US009197520B2

(12) **United States Patent**
**Reynolds et al.**

(10) **Patent No.: US 9,197,520 B2**
(45) **Date of Patent: Nov. 24, 2015**

(54) **METHODS AND COMPUTER PROGRAM PRODUCTS FOR TRANSACTION ANALYSIS OF NETWORK TRAFFIC IN A NETWORK DEVICE**

(71) Applicant: **MICROSOFT TECHNOLOGY LICENSING, LLC**, Redmond, WA (US)

(72) Inventors: **Patrick Alexander Reynolds**, Pensacola, FL (US); **John Branson Bley**, Durham, NC (US); **David William Irwin**, Cary, NC (US); **Aydan R. Yumerefendi**, Raleigh, NC (US); **Glenn Thomas Nethercutt**, Raleigh, NC (US)

(73) Assignee: **Microsoft Technology Licensing, LLC**, Redmond, WA (US)

( * ) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 223 days.

(21) Appl. No.: **13/837,816**

(22) Filed: **Mar. 15, 2013**

(65) **Prior Publication Data**

US 2014/0280892 A1 Sep. 18, 2014

(51) **Int. Cl.**
***H04L 12/26*** (2006.01)
***H04L 12/24*** (2006.01)

(52) **U.S. Cl.**
CPC ............ ***H04L 43/028*** (2013.01); ***H04L 41/069*** (2013.01); *H04L 41/22* (2013.01); *H04L 43/062* (2013.01); *Y04S 40/166* (2013.01); *Y04S 40/168* (2013.01)

(58) **Field of Classification Search**
CPC ..... H04L 43/028; H04L 43/062; H04L 41/22; H04L 41/069; H04L 12/26; Y04S 40/168; Y04S 40/166
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

| | | | | |
|---|---|---|---|---|
| 8,799,225 | B2 * | 8/2014 | Vaitzblit et al. | 707/648 |
| 2002/0087769 | A1 * | 7/2002 | McKenney et al. | 710/200 |
| 2005/0144330 | A1 * | 6/2005 | Richardson | 710/1 |
| 2005/0223368 | A1 * | 10/2005 | Smith et al. | 717/128 |
| 2006/0288149 | A1 * | 12/2006 | LaVigne et al. | 710/306 |
| 2011/0055483 | A1 * | 3/2011 | Heller, Jr. | 711/125 |
| 2012/0304172 | A1 * | 11/2012 | Greifeneder et al. | 718/1 |
| 2014/0032491 | A1 * | 1/2014 | Neerincx et al. | 707/610 |

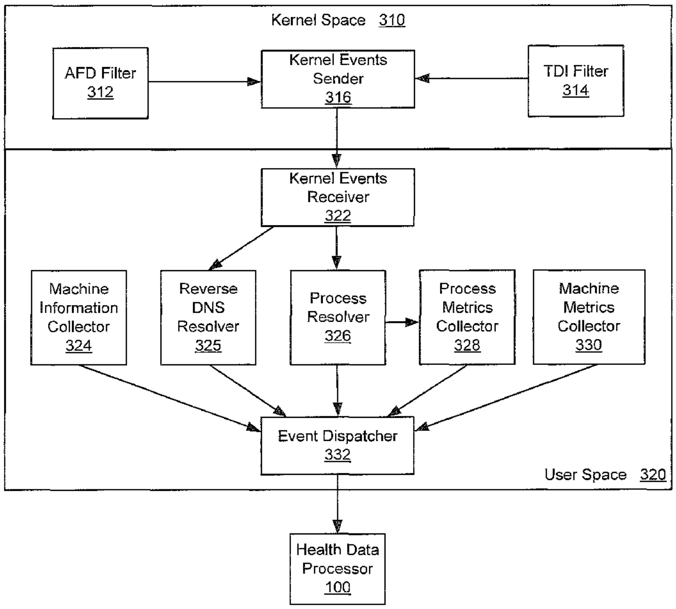
* cited by examiner

*Primary Examiner* — Ebrahim Golabbakhsh
(74) *Attorney, Agent, or Firm* — Henry Gabryjelski; Kate Drakos; Micky Minhas

(57) **ABSTRACT**

Provided are methods and computer program products for analyzing and tracing the contents of network traffic in a network device and to help identify network performance issues. Methods for analyzing traffic include identifying thread identifiers associated with transactions, determining related events, and associating similar transactions and events into an aggregated list of transactions. Methods of tracing transactions include identifying thread identifiers associated with transactions and related events and serializing events into a serialized list of events to facilitate tracing single transactions.

**34 Claims, 35 Drawing Sheets**



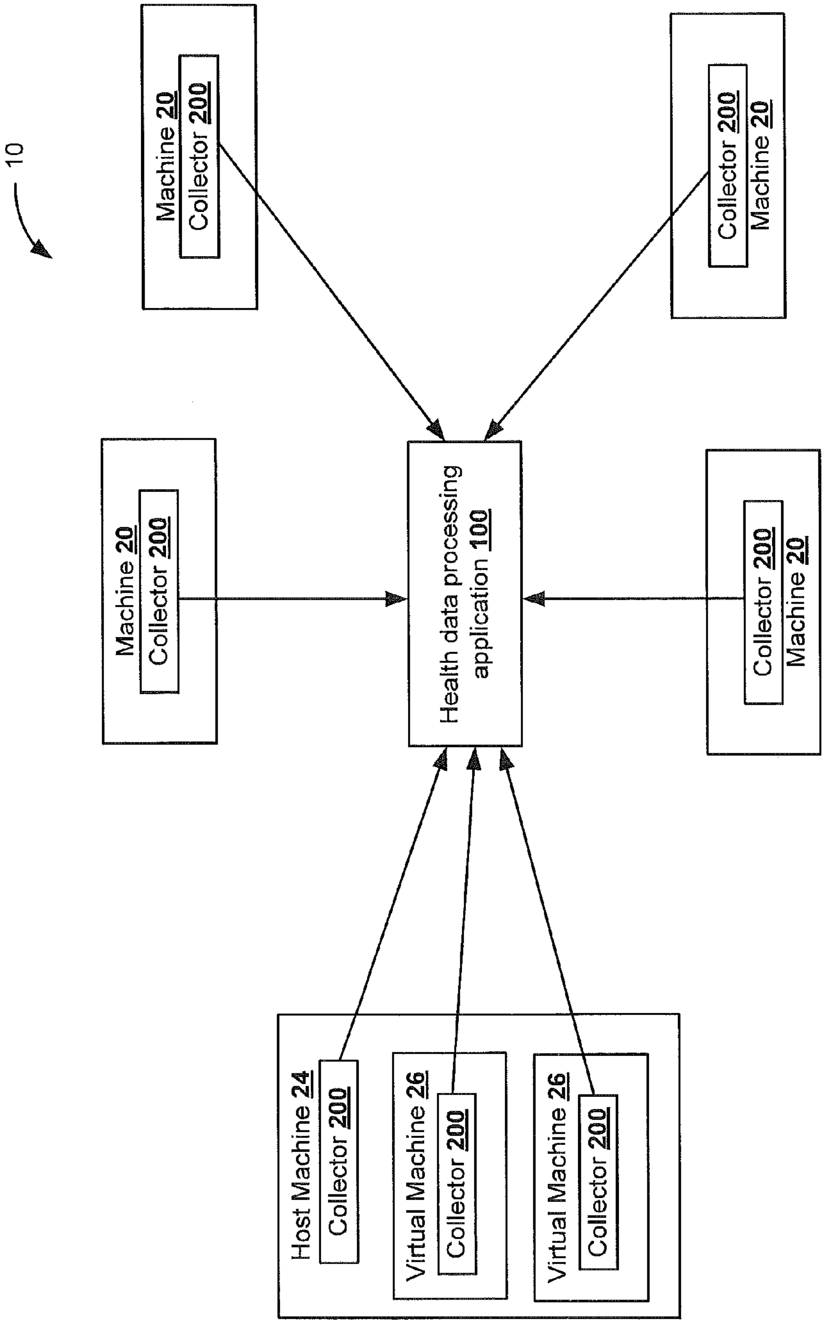

10
Machine 20
Collector 200
Collector 200
Machine 20
Machine 20
Collector 200
Health data processing application 100
Collector 200
Machine 20
Host Machine 24
Collector 200
Virtual Machine 26
Collector 200
Virtual Machine 26
Collector 200

FIG. 1a

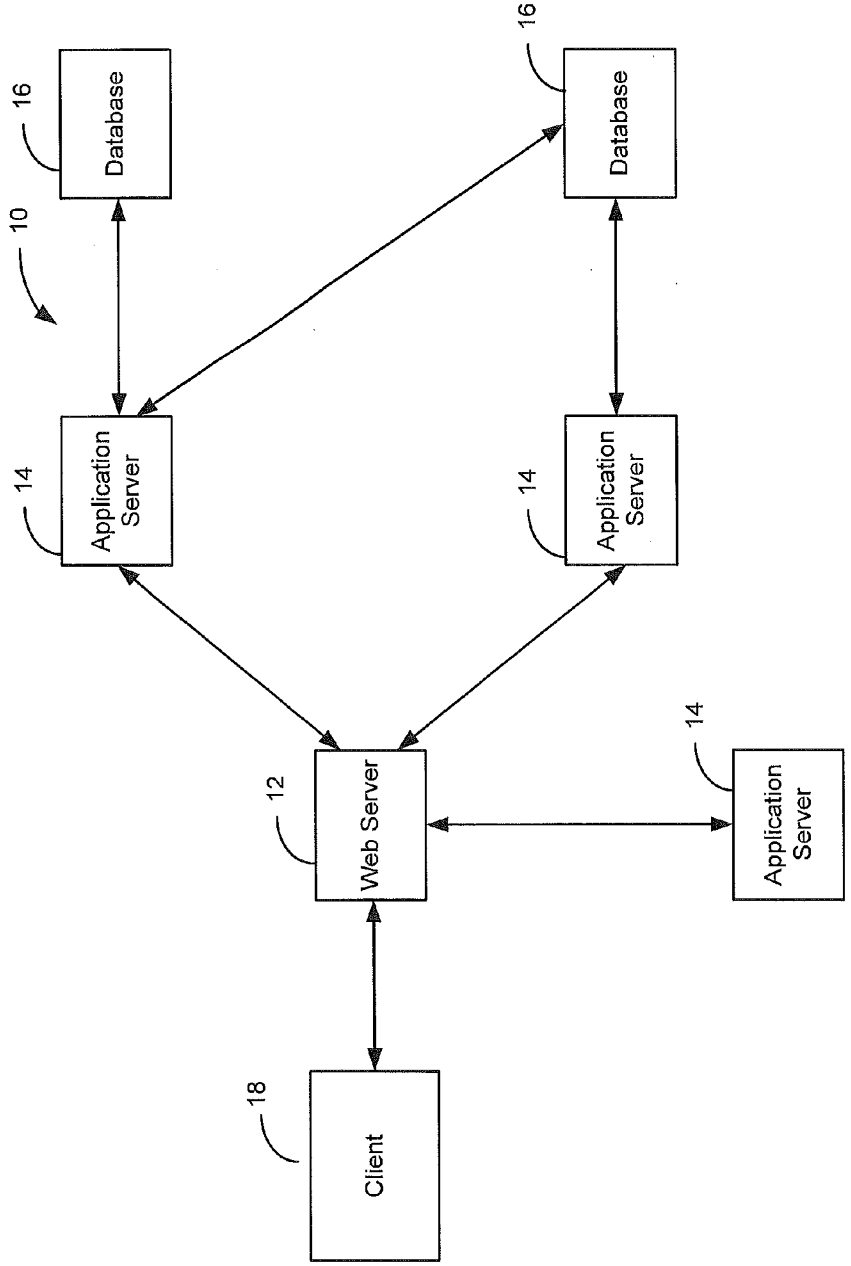

10
16
Database
16
Database
14
Application Server
14
Application Server
12
Web Server
14
Application Server
18
Client

FIG. 1b

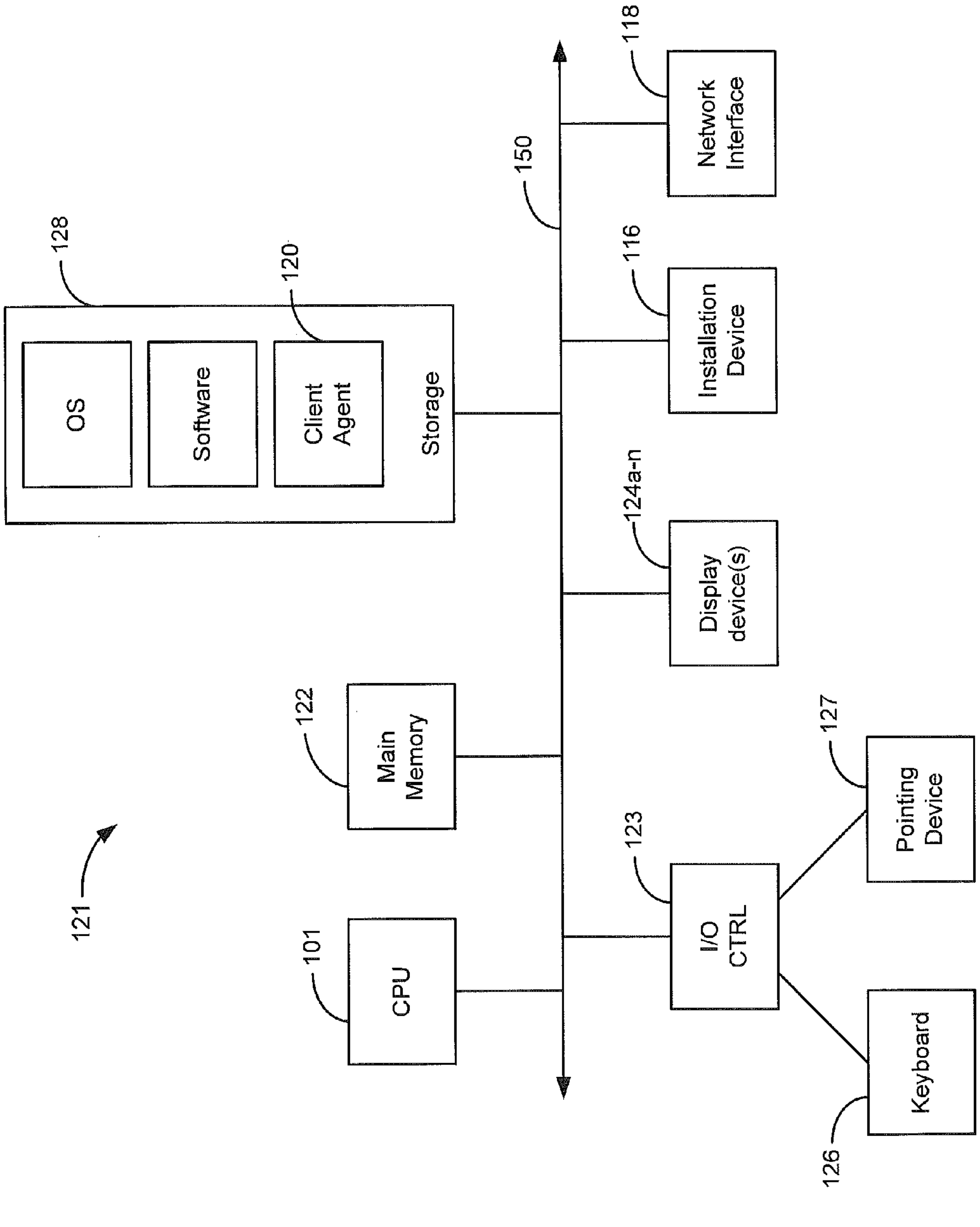

121
128
OS
Software
Client Agent
120
Storage
150
118
Network Interface
116
Installation Device
124a-n
Display device(s)
122
Main Memory
101
CPU
123
I/O CTRL
127
Pointing Device
126
Keyboard

FIG. 1c

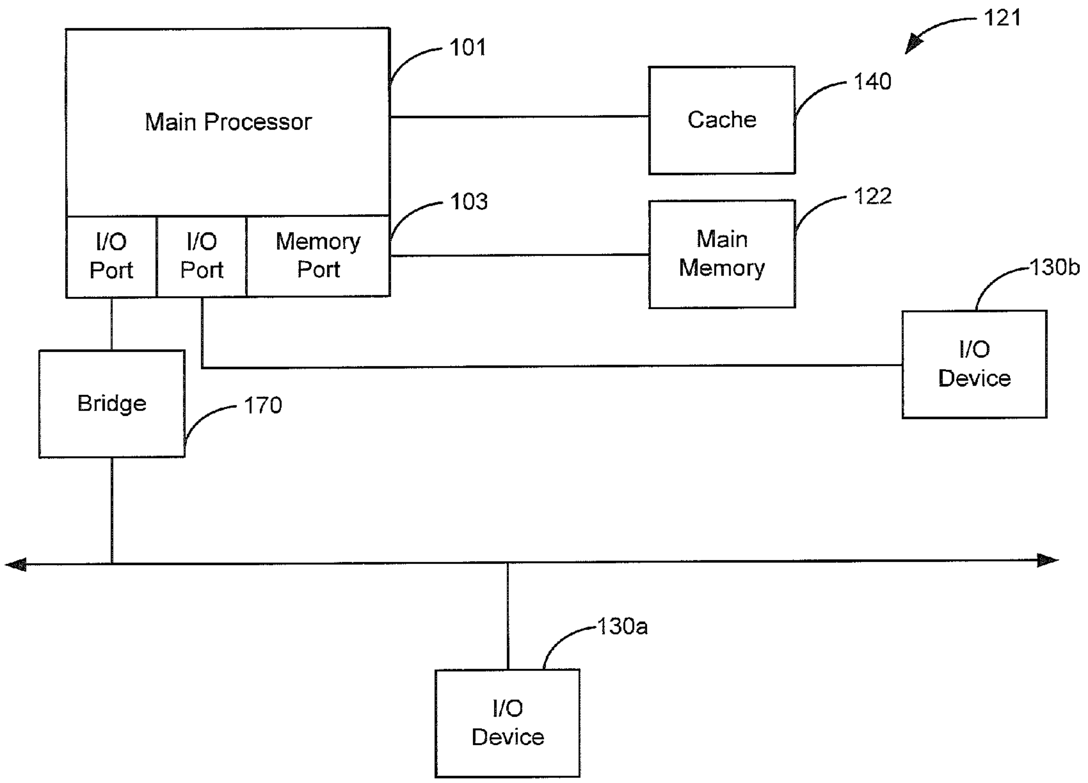

121
Main Processor
101
Cache
140
I/O Port
I/O Port
Memory Port
103
Main Memory
122
130b
I/O Device
Bridge
170
130a
I/O Device

FIG. 1d

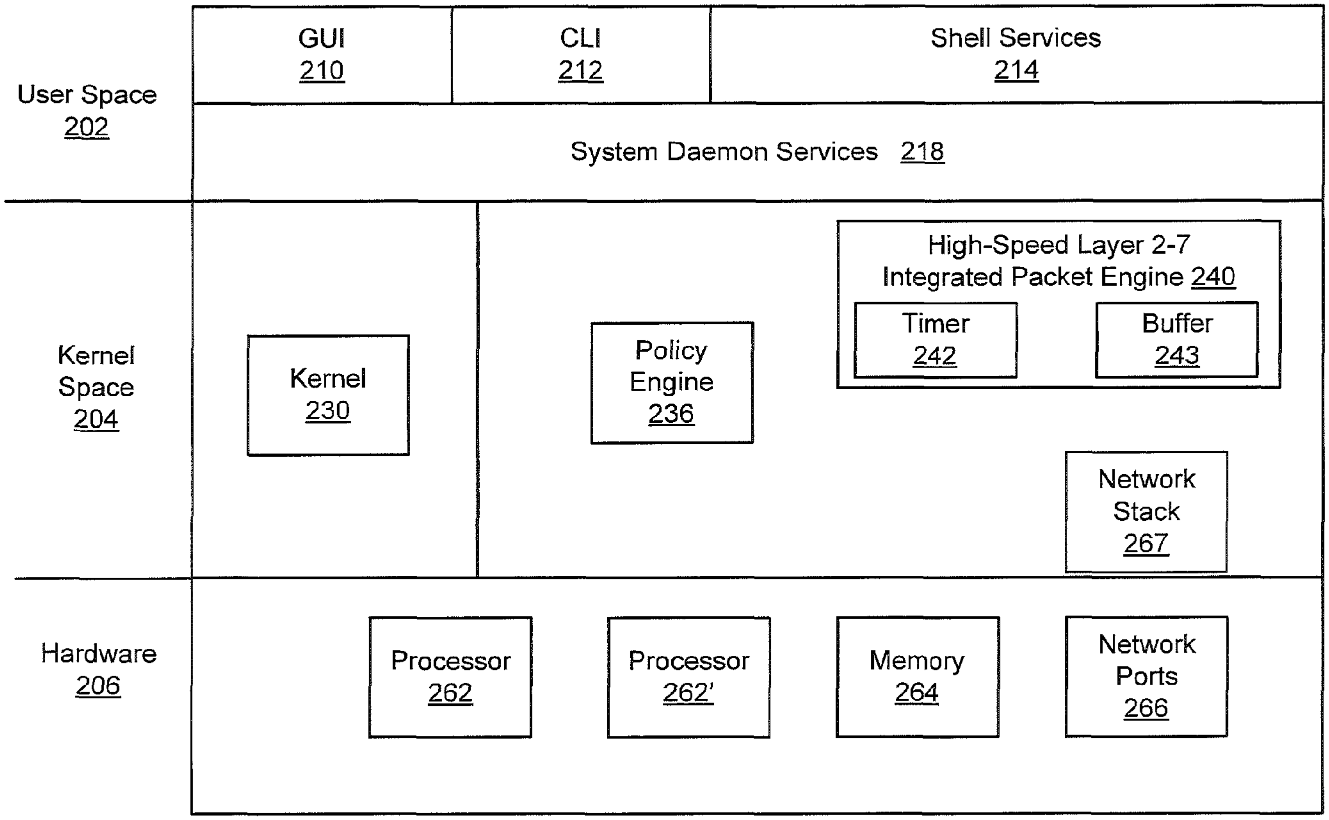

User Space
202
Kernel
Space
204
Hardware
206
GUI
210
CLI
212
Shell Services
214
System Daemon Services 218
Kernel
230
Policy
Engine
236
High-Speed Layer 2-7
Integrated Packet Engine 240
Timer
242
Buffer
243
Network
Stack
267
Processor
262
Processor
262'
Memory
264
Network
Ports
266

FIG. 2

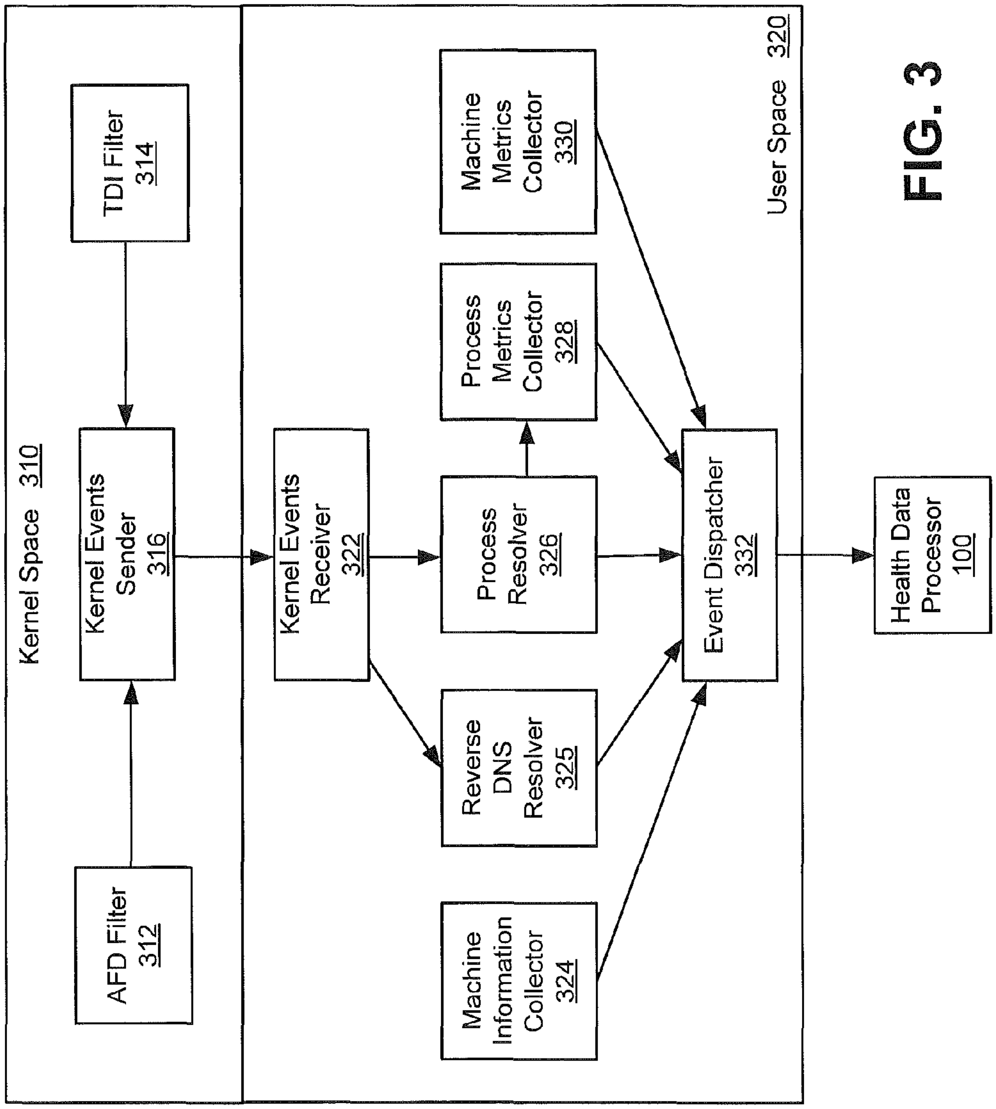

Kernel Space 310
TDI Filter 314
Kernel Events Sender 316
AFD Filter 312
User Space 320
Machine Metrics Collector 330
Process Metrics Collector 328
Kernel Events Receiver 322
Process Resolver 326
Reverse DNS Resolver 325
Machine Information Collector 324
Event Dispatcher 332
Health Data Processor 100

FIG. 3

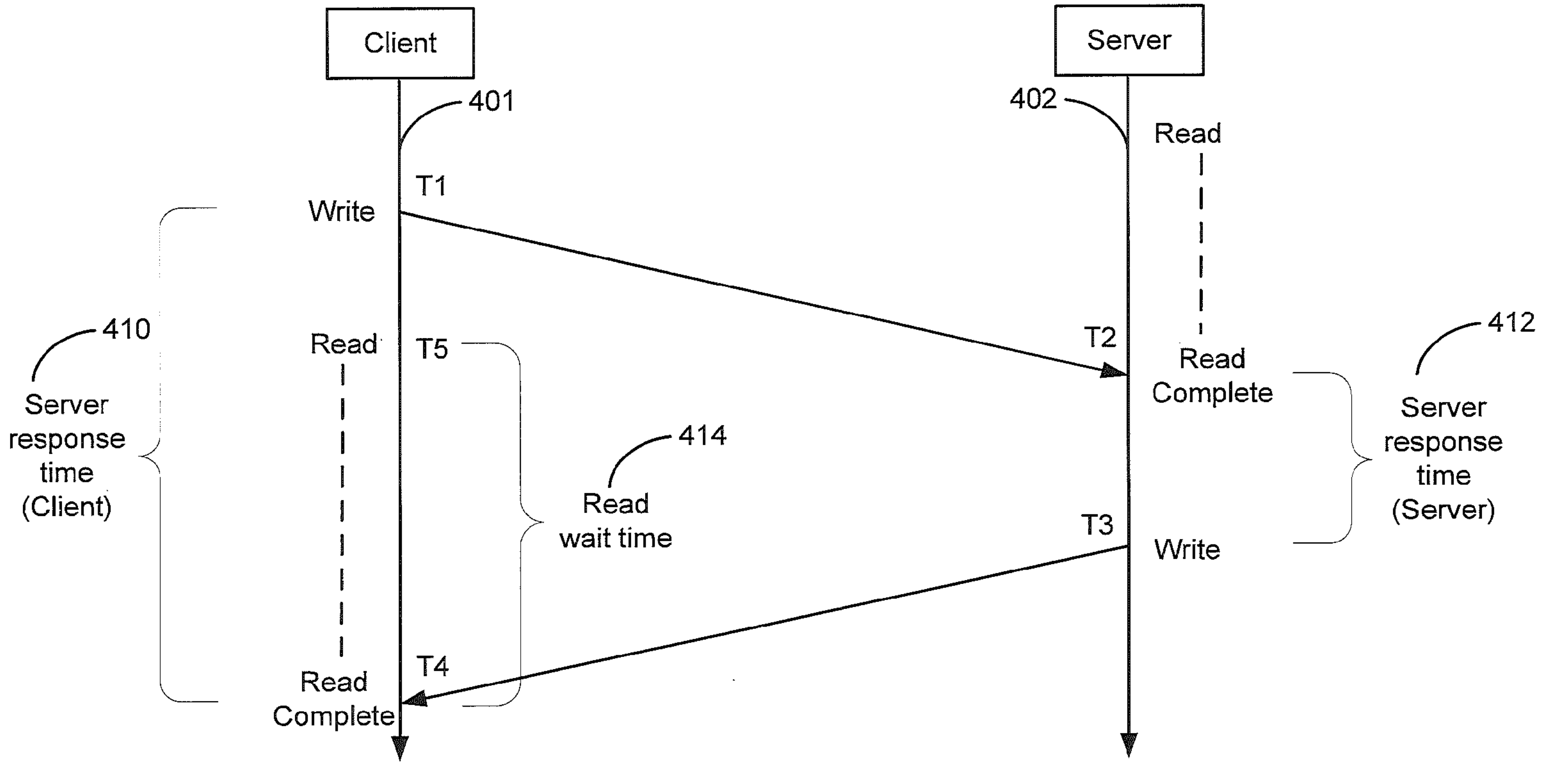

Client
Server
401
402
Read
T1
Write
410
Read
T5
T2
Read
Complete
412
Server
response
time
(Client)
414
Read
wait time
Server
response
time
(Server)
T3
Write
T4
Read
Complete

FIG. 4

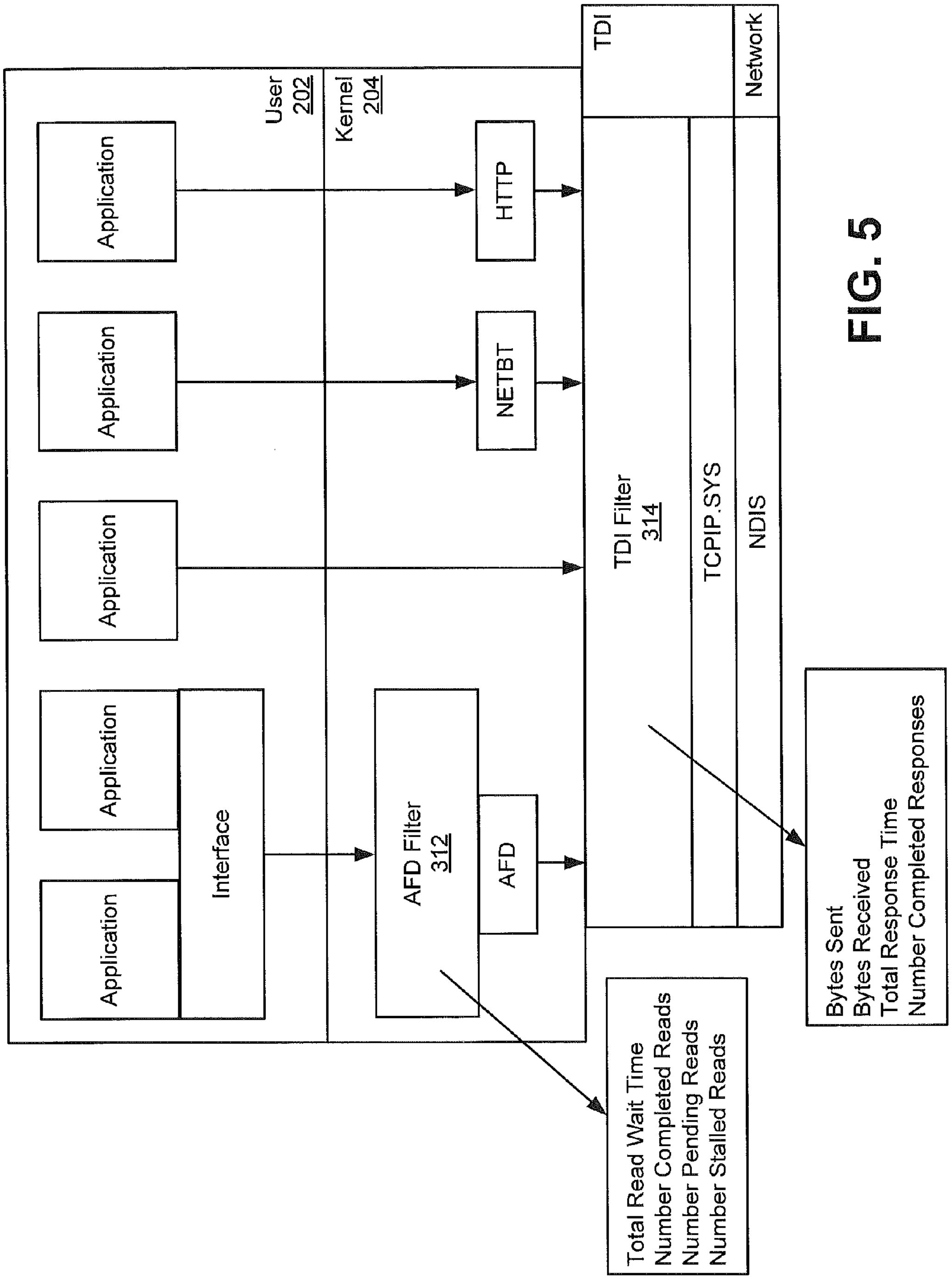

User 202
Kernel 204
Application
Application
Application
Application
Application
Interface
AFD Filter 312
AFD
NETBT
HTTP
TDI
TDI Filter 314
TCPIP.SYS
Network
NDIS
Total Read Wait Time
Number Completed Reads
Number Pending Reads
Number Stalled Reads
Bytes Sent
Bytes Received
Total Response Time
Number Completed Responses

FIG. 5

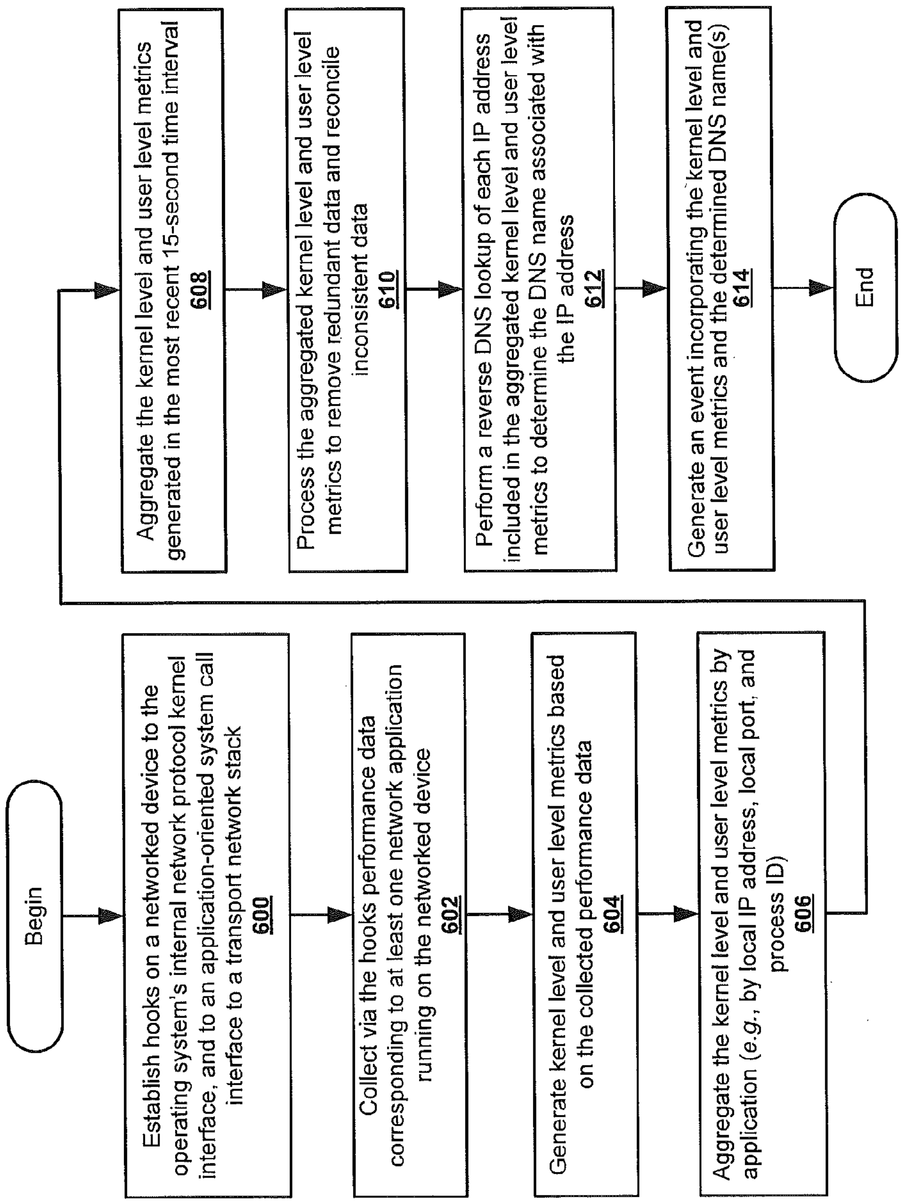

Begin
Establish hooks on a networked device to the operating system's internal network protocol kernel interface, and to an application-oriented system call interface to a transport network stack
600
Collect via the hooks performance data corresponding to at least one network application running on the networked device
602
Generate kernel level and user level metrics based on the collected performance data
604
Aggregate the kernel level and user level metrics by application (e.g., by local IP address, local port, and process ID)
606
Aggregate the kernel level and user level metrics generated in the most recent 15-second time interval
608
Process the aggregated kernel level and user level metrics to remove redundant data and reconcile inconsistent data
610
Perform a reverse DNS lookup of each IP address included in the aggregated kernel level and user level metrics to determine the DNS name associated with the IP address
612
Generate an event incorporating the kernel level and user level metrics and the determined DNS name(s)
614
End

FIG. 6

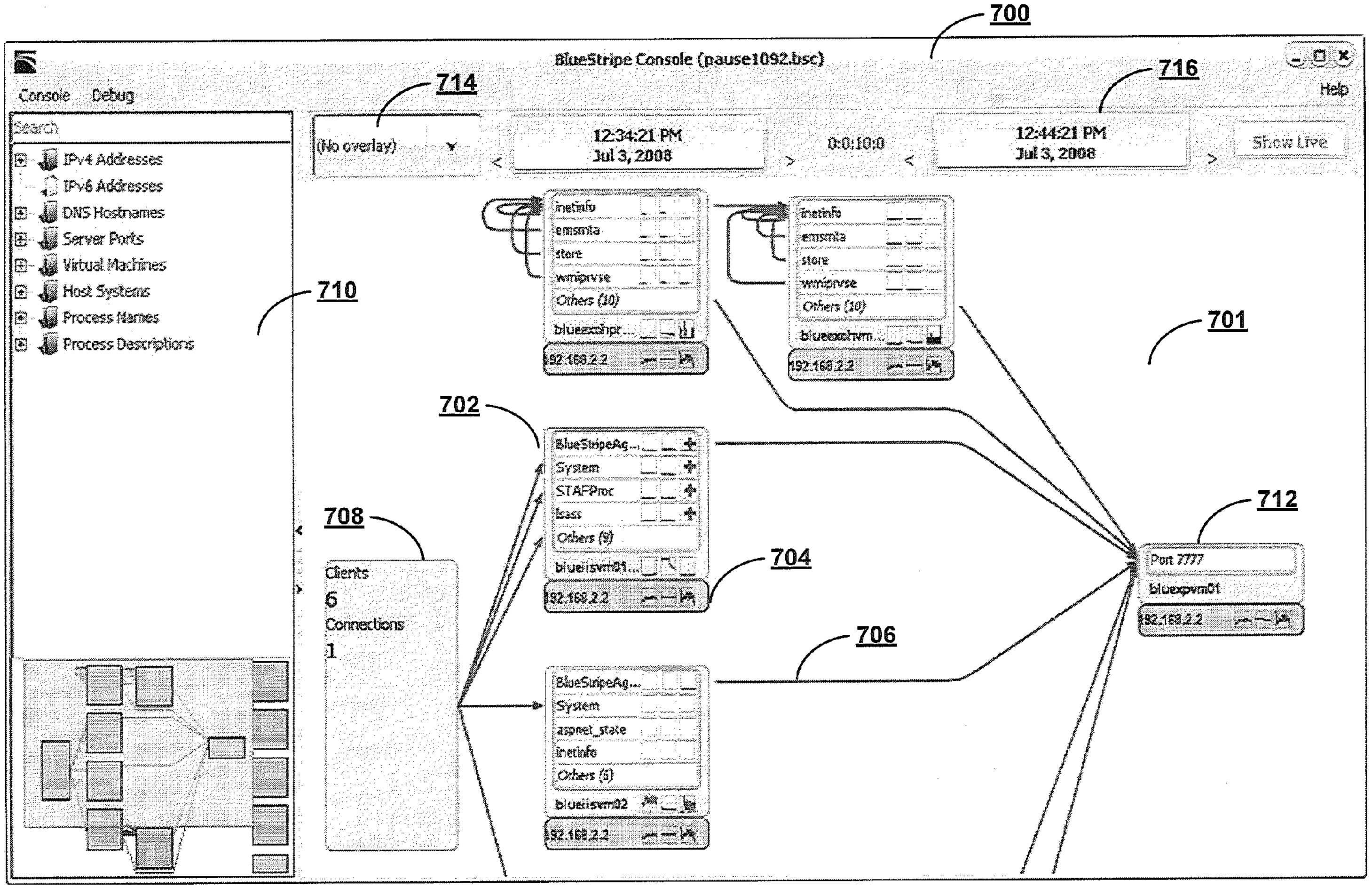

700
BlueStripe Console (pause1092.bsc)
Console
Debug
Help
714
(No overlay)
12:34:21 PM
Jul 3, 2008
0:0:10:0
716
12:44:21 PM
Jul 3, 2008
Show Live
Search
IPv4 Addresses
IPv6 Addresses
DNS Hostnames
Server Ports
Virtual Machines
Host Systems
Process Names
Process Descriptions
710
inetinfo
emsmta
store
wmiprvse
Others (10)
blueexchpr...
192.168.2.2
inetinfo
emsmta
store
wmiprvse
Others (10)
blueexchvm...
192.168.2.2
701
702
BlueStripeAg...
System
STAFProc
lsass
Others (9)
blueiisvm01...
192.168.2.2
704
708
Clients
6
Connections
1
712
Port 7777
bluexpvm01
192.168.2.2
706
BlueStripeAg...
System
aspnet_state
inetinfo
Others (5)
blueiisvm02
192.168.2.2

FIG. 7

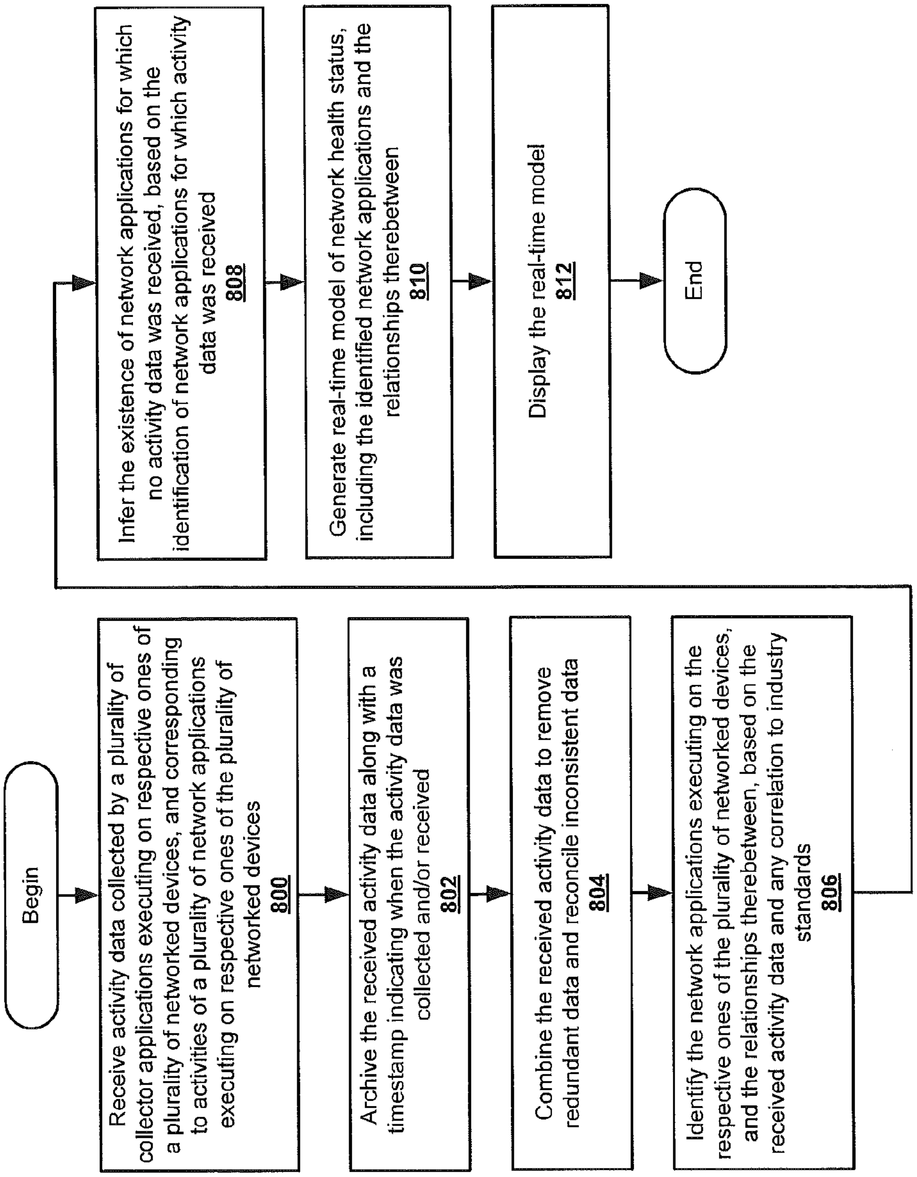

Begin
Receive activity data collected by a plurality of collector applications executing on respective ones of a plurality of networked devices, and corresponding to activities of a plurality of network applications executing on respective ones of the plurality of networked devices
800
Archive the received activity data along with a timestamp indicating when the activity data was collected and/or received
802
Combine the received activity data to remove redundant data and reconcile inconsistent data
804
Identify the network applications executing on the respective ones of the plurality of networked devices, and the relationships therebetween, based on the received activity data and any correlation to industry standards
806
Infer the existence of network applications for which no activity data was received, based on the identification of network applications for which activity data was received
808
Generate real-time model of network health status, including the identified network applications and the relationships therebetween
810
Display the real-time model
812
End

FIG. 8

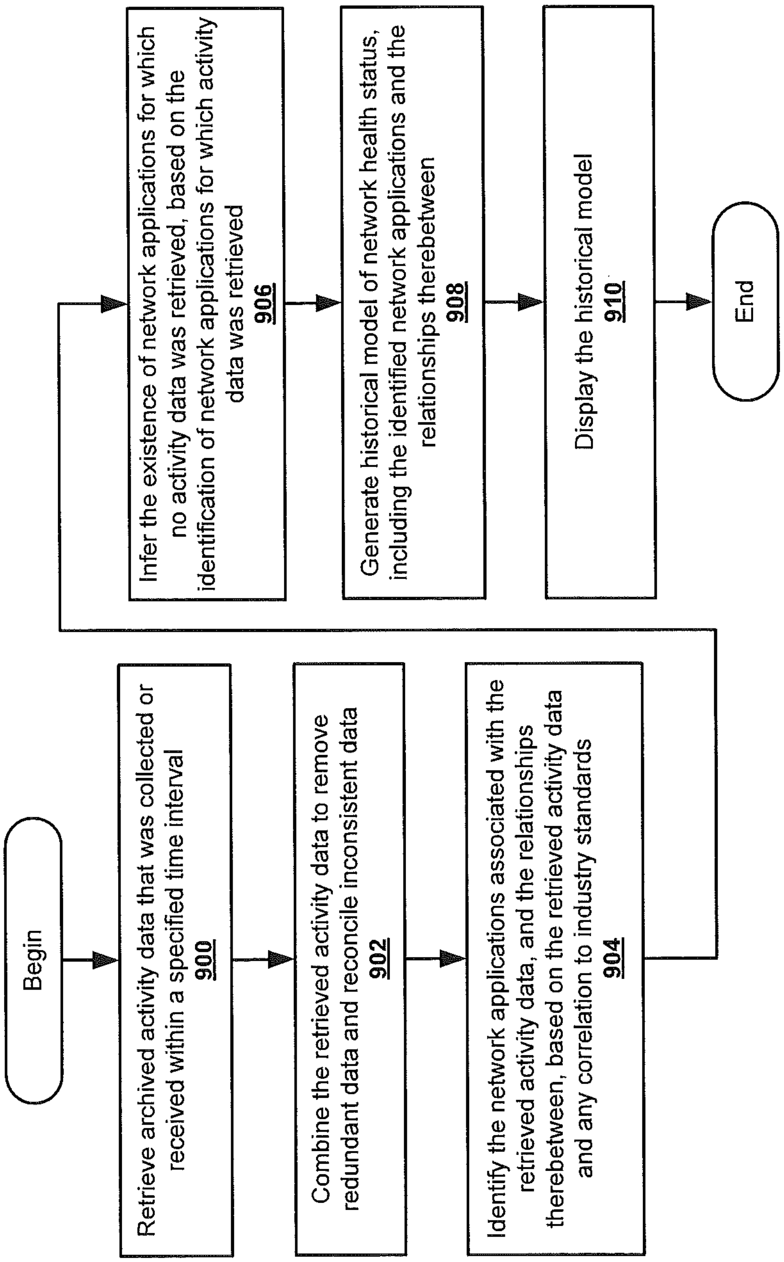

Begin
Retrieve archived activity data that was collected or received within a specified time interval
900
Combine the retrieved activity data to remove redundant data and reconcile inconsistent data
902
Identify the network applications associated with the retrieved activity data, and the relationships therebetween, based on the retrieved activity data and any correlation to industry standards
904
Infer the existence of network applications for which no activity data was retrieved, based on the identification of network applications for which activity data was retrieved
906
Generate historical model of network health status, including the identified network applications and the relationships therebetween
908
Display the historical model
910
End

FIG. 9

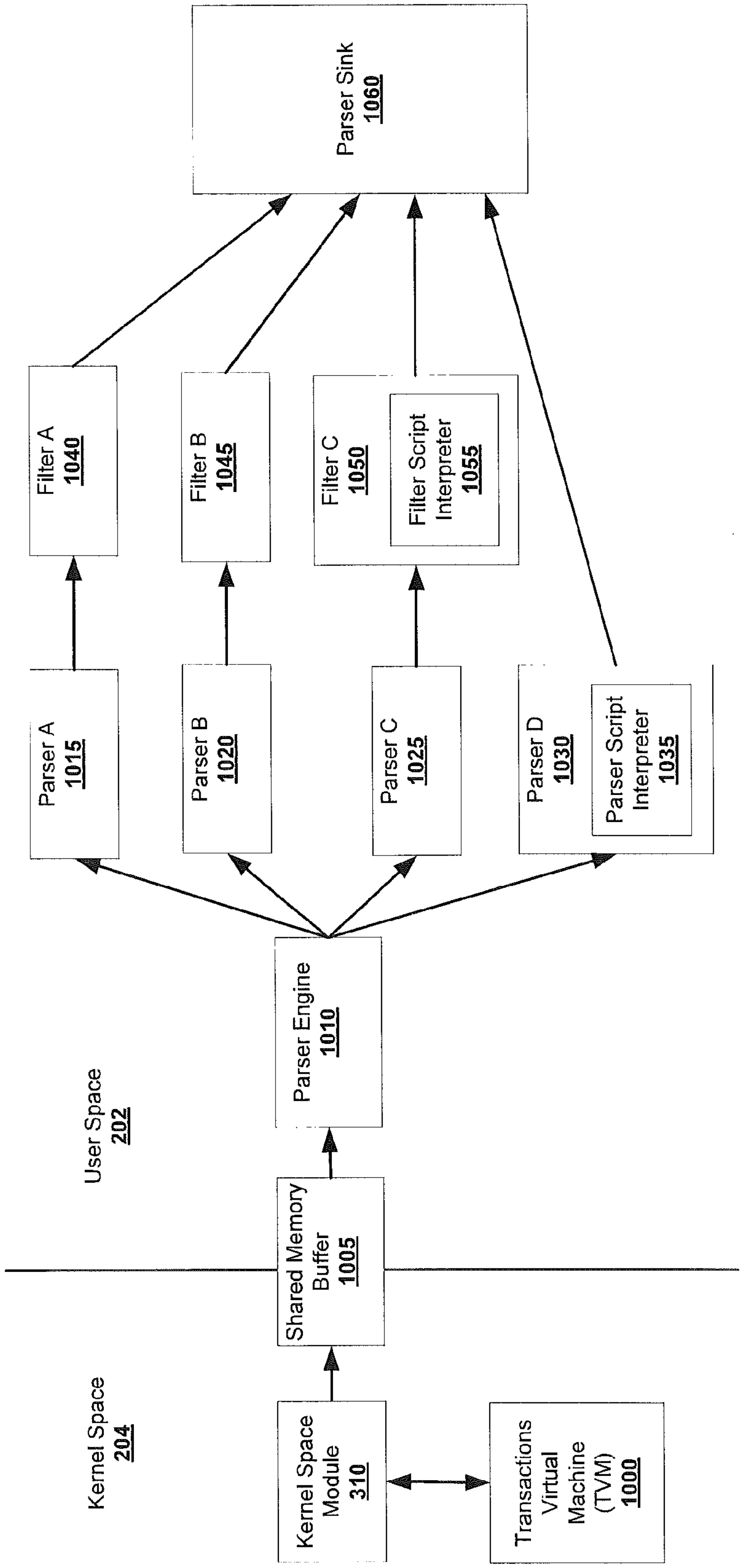

Kernel Space
204
User Space
202
Kernel Space Module
310
Transactions Virtual Machine (TVM)
1000
Shared Memory Buffer
1005
Parser Engine
1010
Parser A
1015
Parser B
1020
Parser C
1025
Parser D
1030
Parser Script Interpreter
1035
Filter A
1040
Filter B
1045
Filter C
1050
Filter Script Interpreter
1055
Parser Sink
1060

FIG. 10

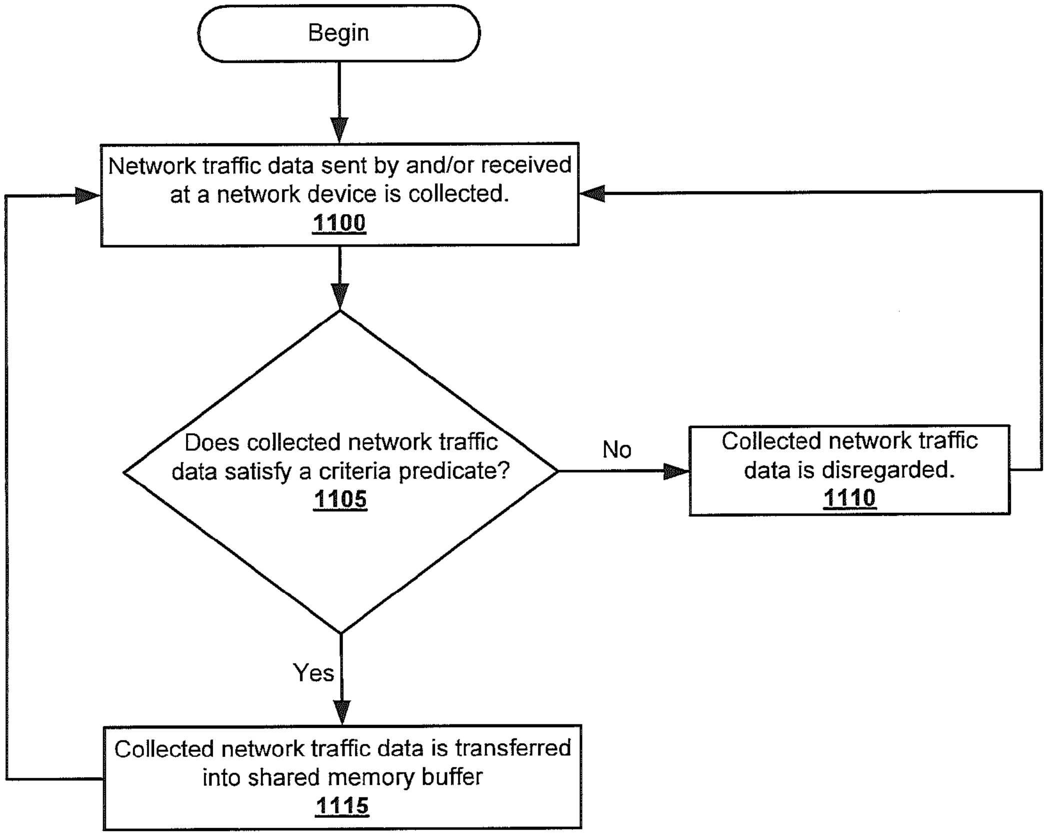

Begin
Network traffic data sent by and/or received at a network device is collected.
1100
Does collected network traffic data satisfy a criteria predicate?
1105
No
Collected network traffic data is disregarded.
1110
Yes
Collected network traffic data is transferred into shared memory buffer
1115

FIG. 11

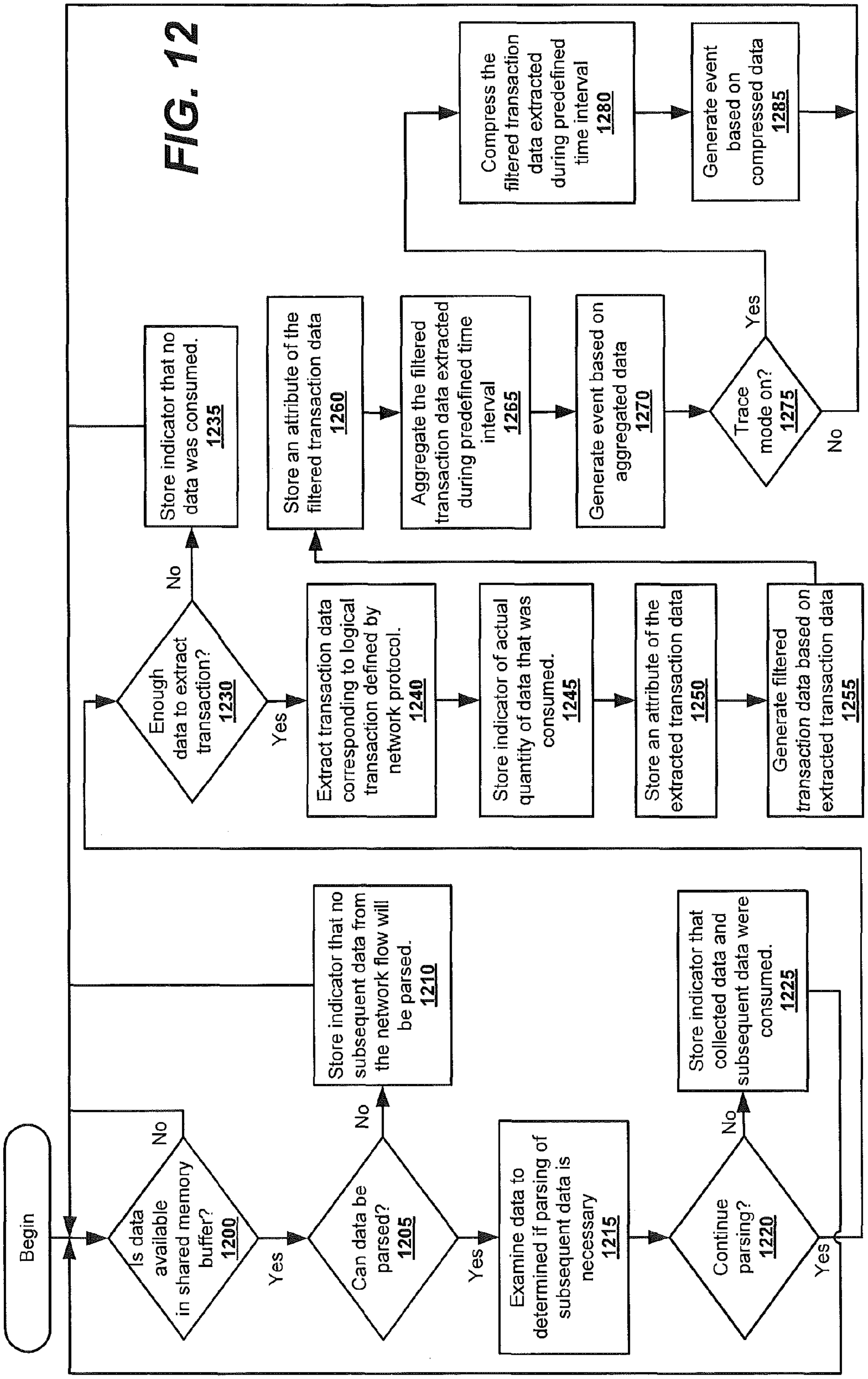

Begin
Is data available in shared memory buffer?
1200
No
Yes
Can data be parsed?
1205
No
Yes
Store indicator that no subsequent data from the network flow will be parsed.
1210
Examine data to determined if parsing of subsequent data is necessary
1215
Continue parsing?
1220
No
Yes
Store indicator that collected data and subsequent data were consumed.
1225
Enough data to extract transaction?
1230
No
Yes
Store indicator that no data was consumed.
1235
Extract transaction data corresponding to logical transaction defined by network protocol.
1240
Store indicator of actual quantity of data that was consumed.
1245
Store an attribute of the extracted transaction data
1250
Generate filtered transaction data based on extracted transaction data
1255
Store an attribute of the filtered transaction data
1260
Aggregate the filtered transaction data extracted during predefined time interval
1265
Generate event based on aggregated data
1270
Trace mode on?
1275
Yes
No
Compress the filtered transaction data extracted during predefined time interval
1280
Generate event based on compressed data
1285

FIG. 12

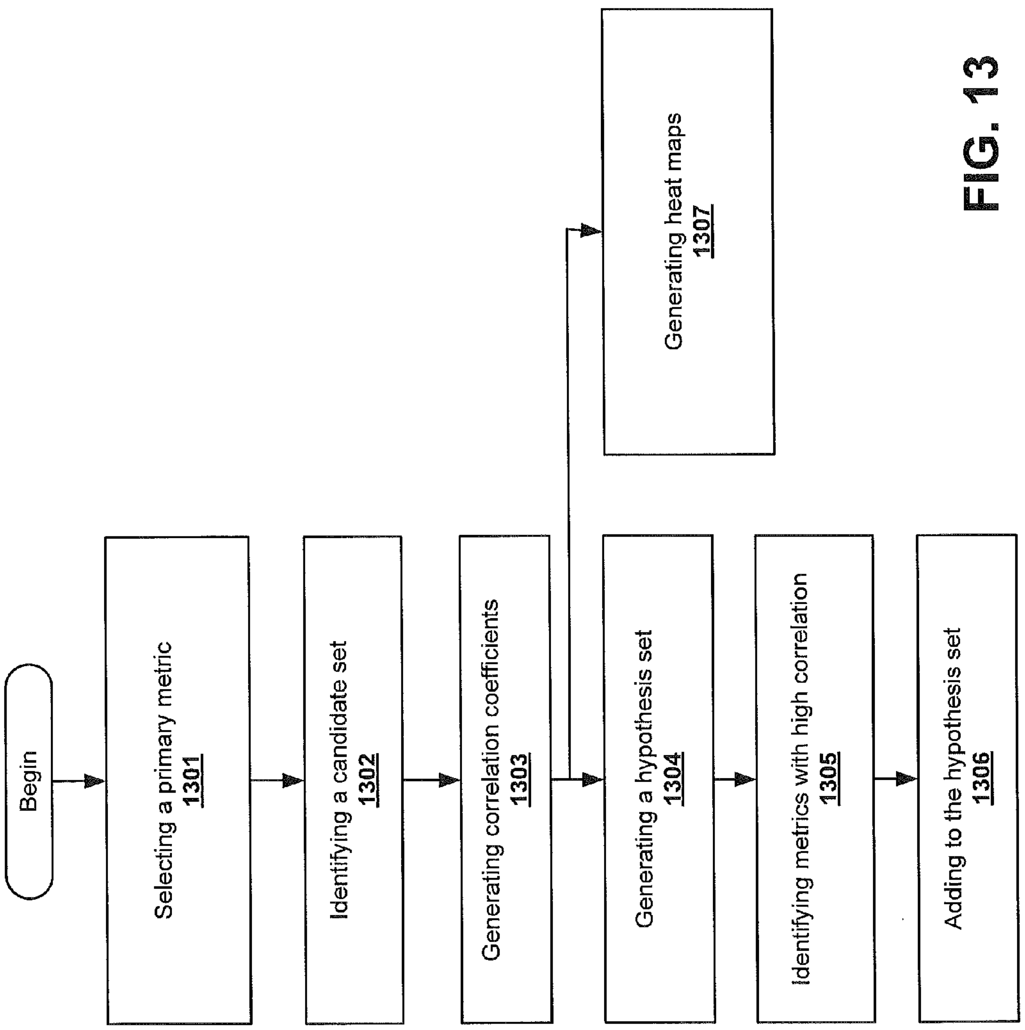

Begin
Selecting a primary metric
1301
Identifying a candidate set
1302
Generating correlation coefficients
1303
Generating a hypothesis set
1304
Identifying metrics with high correlation
1305
Adding to the hypothesis set
1306
Generating heat maps
1307

FIG. 13

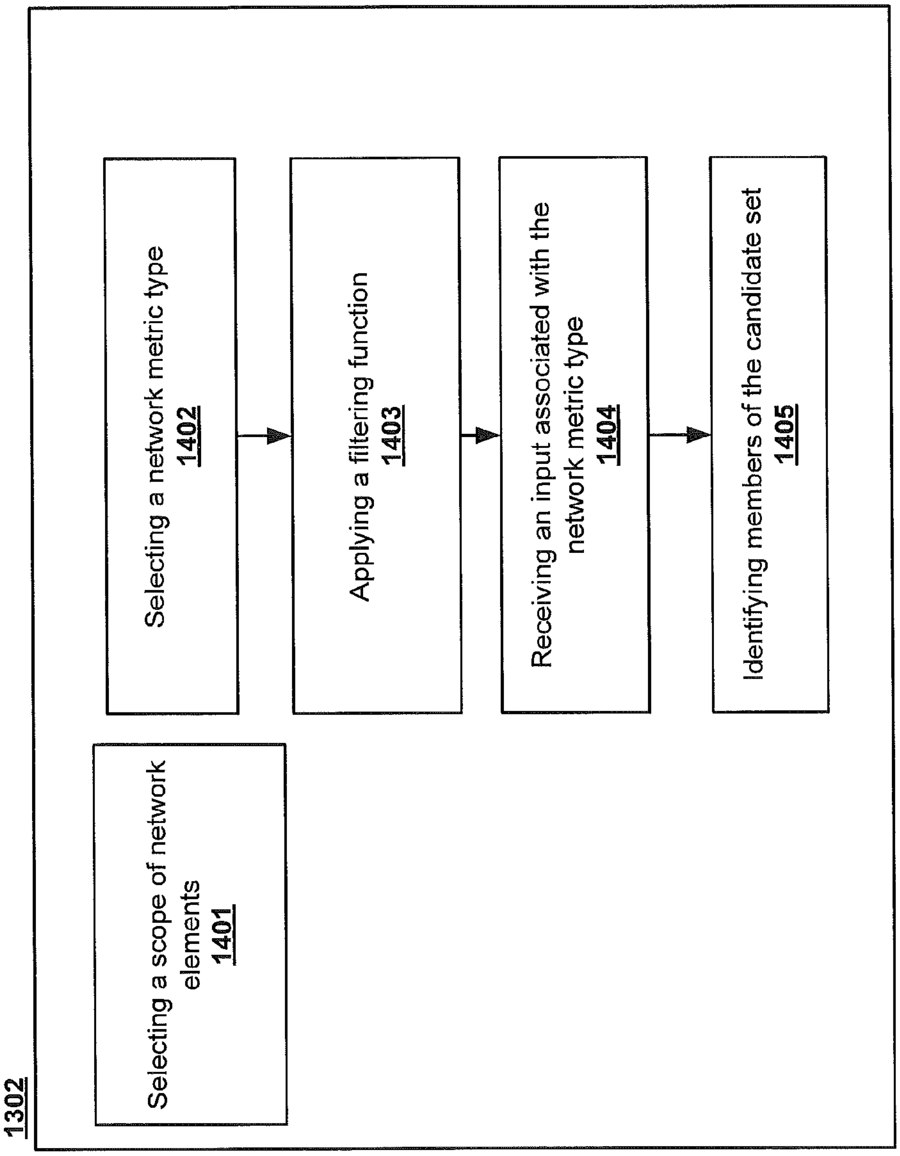

1302
Selecting a scope of network elements
1401
Selecting a network metric type
1402
Applying a filtering function
1403
Receiving an input associated with the network metric type
1404
Identifying members of the candidate set
1405

FIG. 14

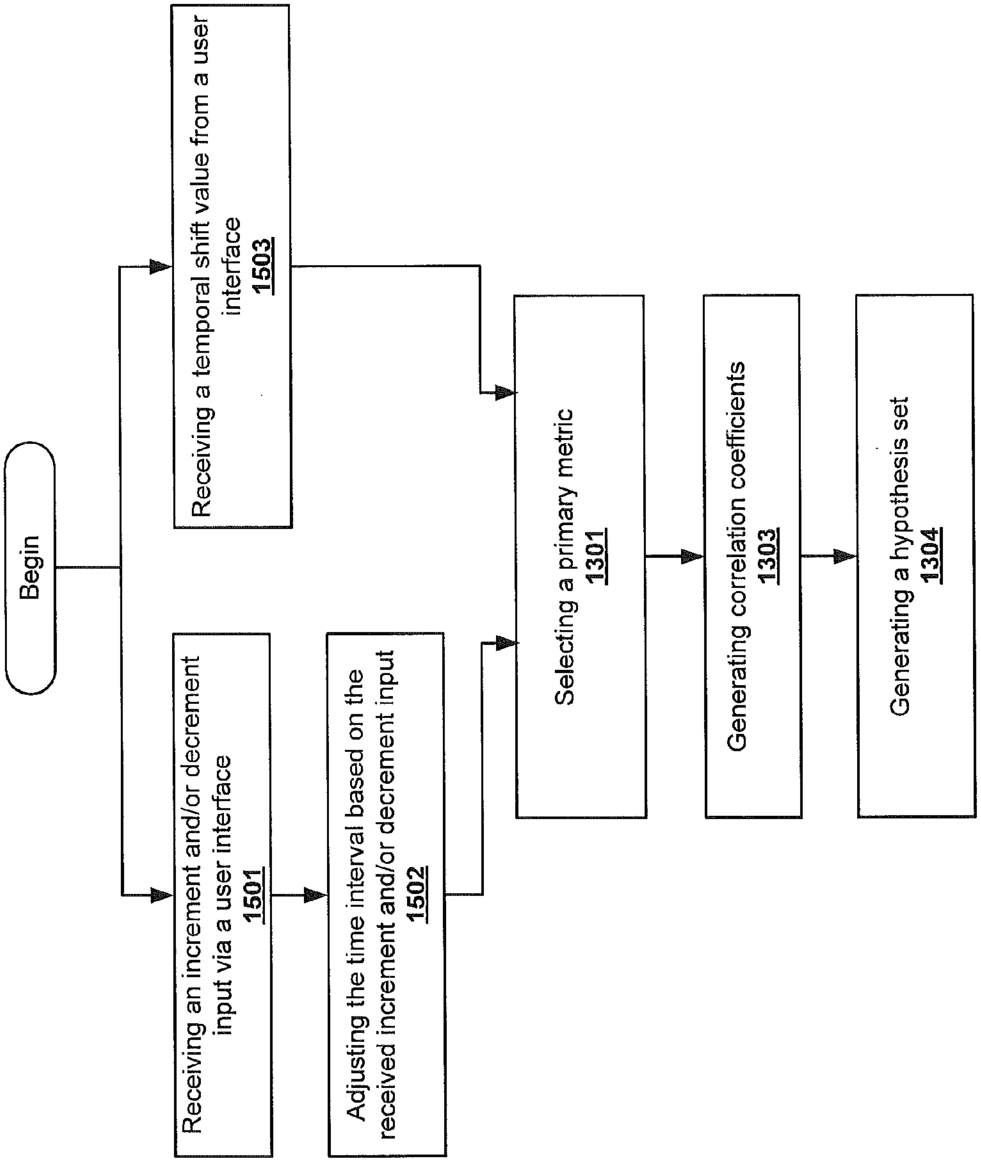

Begin
Receiving a temporal shift value from a user interface
1503
Receiving an increment and/or decrement input via a user interface
1501
Adjusting the time interval based on the received increment and/or decrement input
1502
Selecting a primary metric
1301
Generating correlation coefficients
1303
Generating a hypothesis set
1304

FIG. 15

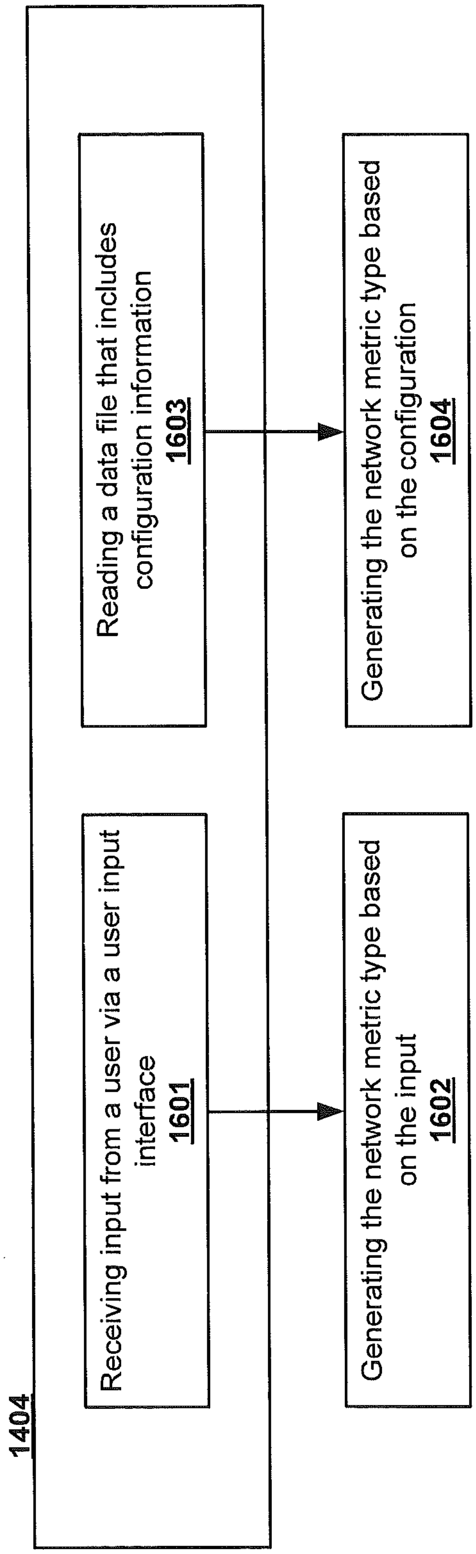

1404
Receiving input from a user via a user input interface
1601
Generating the network metric type based on the input
1602
Reading a data file that includes configuration information
1603
Generating the network metric type based on the configuration
1604

FIG. 16

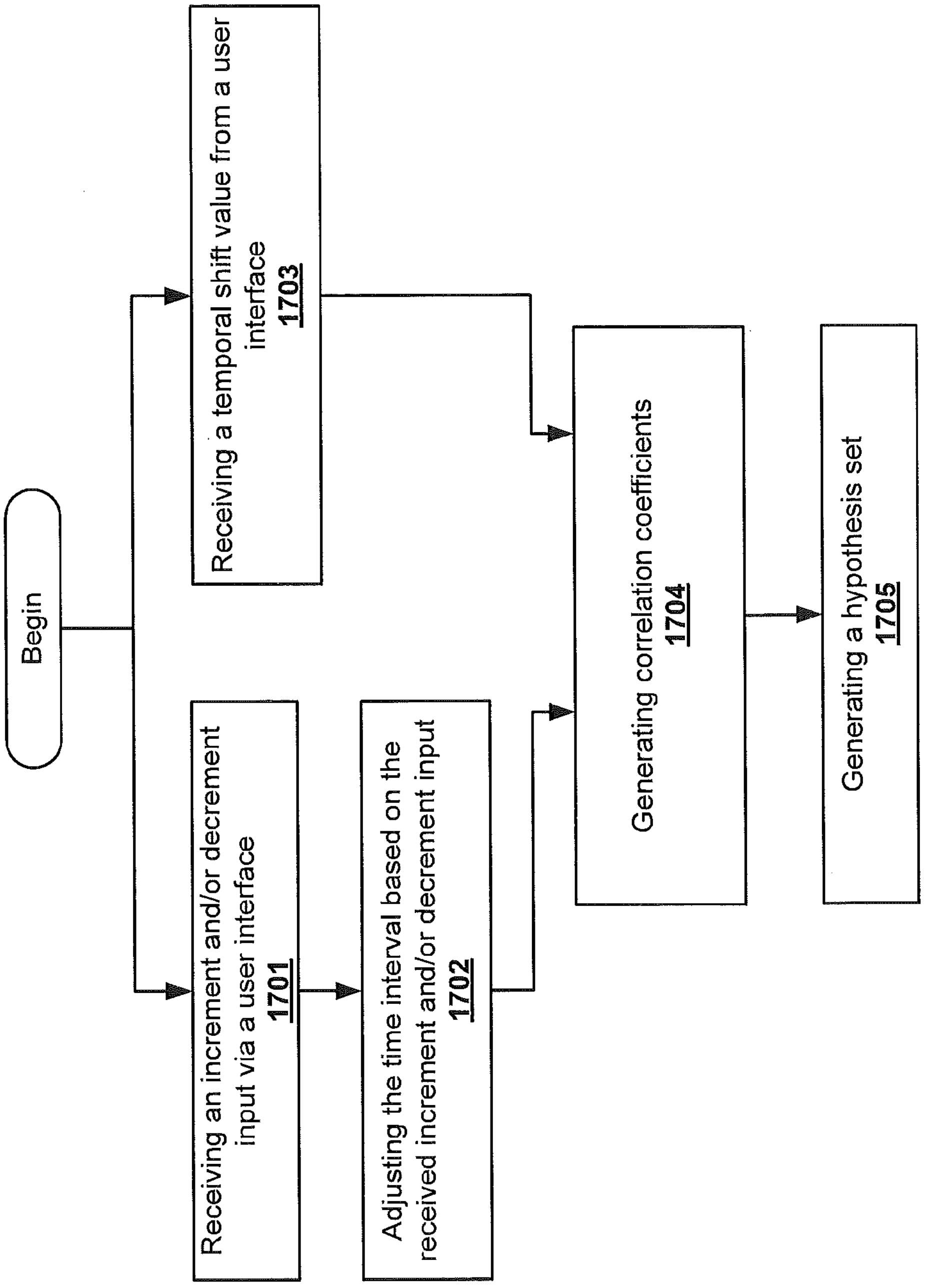
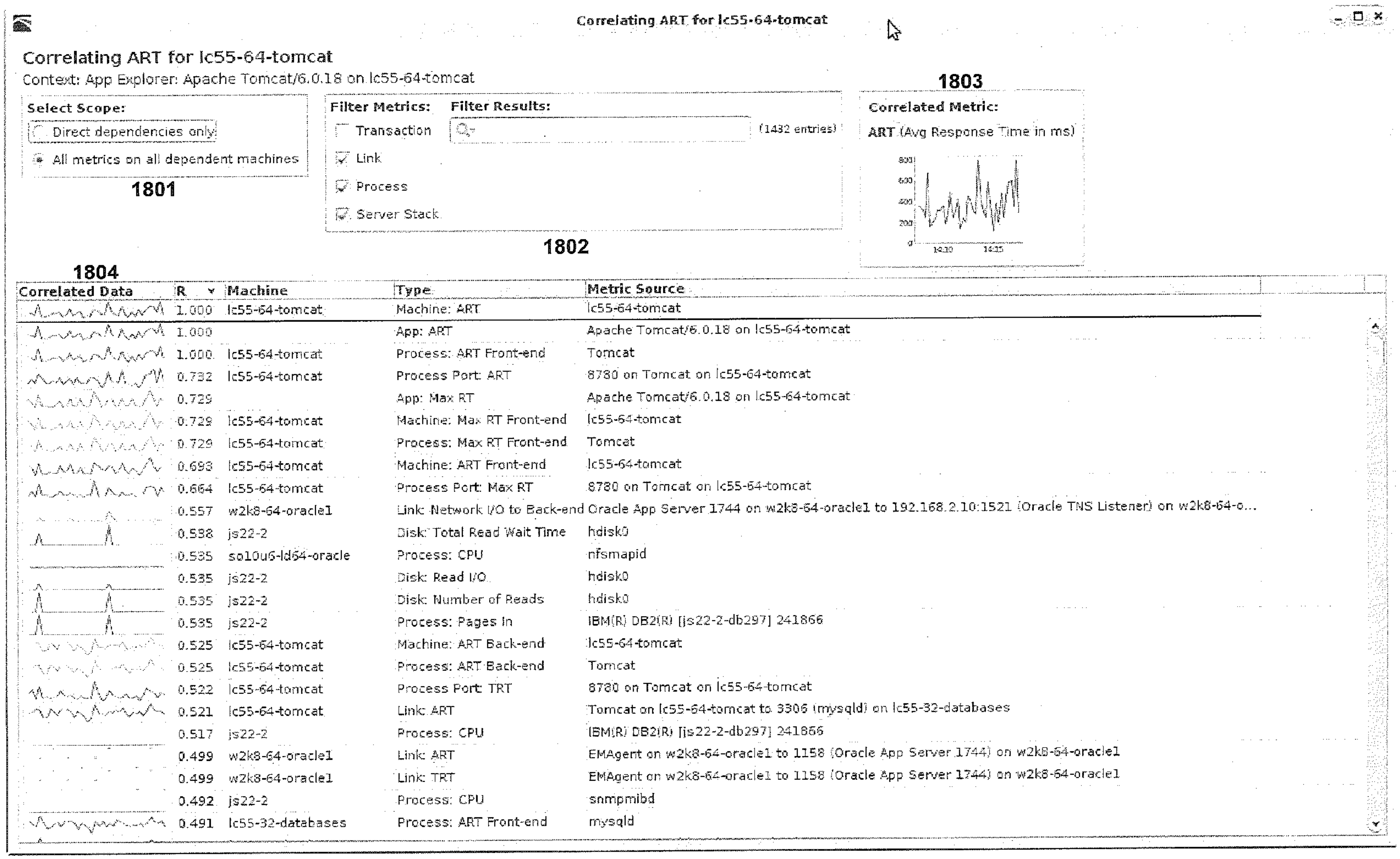

Begin
Receiving an increment and/or decrement input via a user interface
1701
Adjusting the time interval based on the received increment and/or decrement input
1702
Receiving a temporal shift value from a user interface
1703
Generating correlation coefficients
1704
Generating a hypothesis set
1705

FIG. 17



Correlating ART for lc55-64-tomcat

Correlating ART for lc55-64-tomcat

Context: App Explorer: Apache Tomcat/6.0.18 on lc55-64-tomcat

Select Scope:

○ Direct dependencies only

● All metrics on all dependent machines

Filter Metrics:

□ Transaction

☑ Link

☑ Process

☑ Server Stack

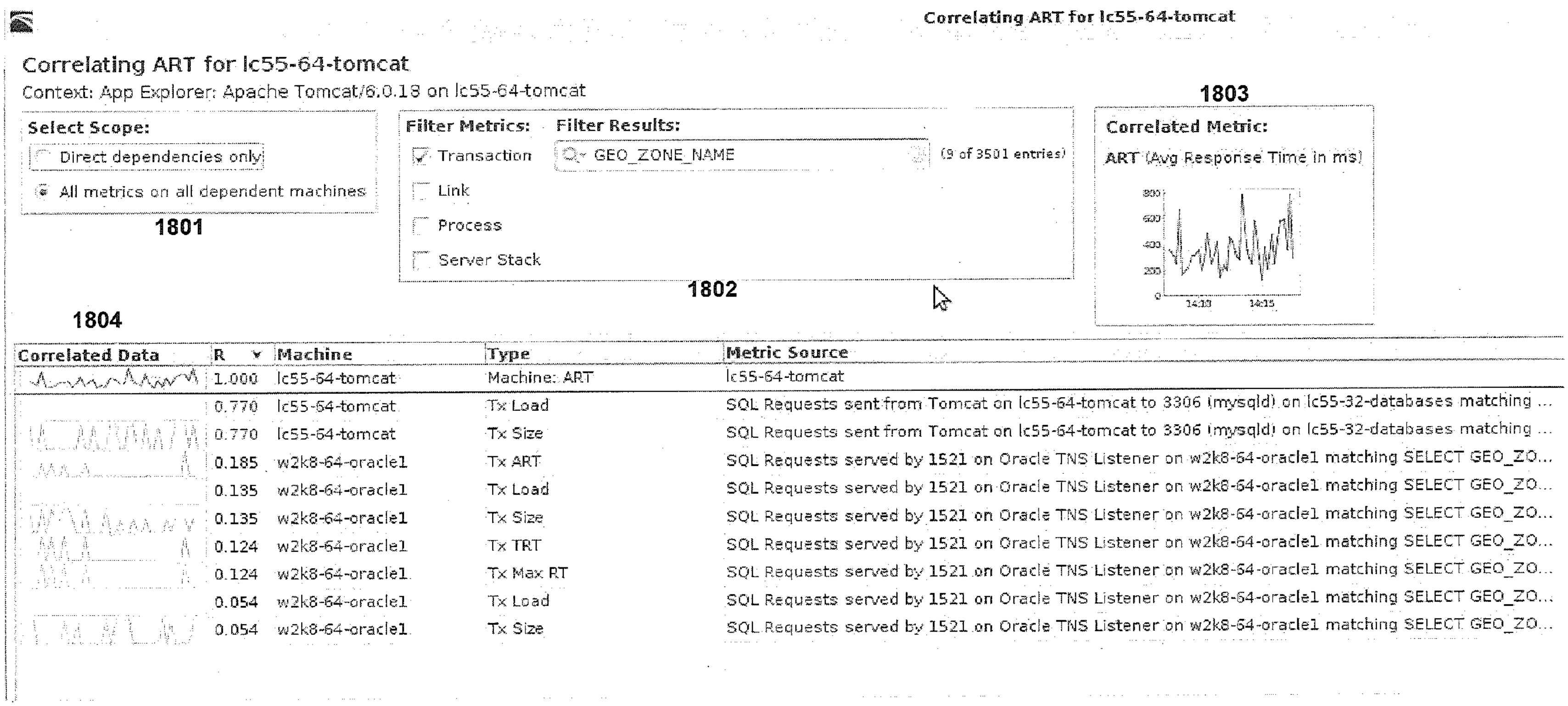
Filter Results: (1432 entries)

Correlated Metric:

ART (Avg Response Time in ms)

| Correlated Data | R | Machine | Type | Metric Source |
|---|---|---|---|---|
| | 1.000 | lc55-64-tomcat | Machine: ART | lc55-64-tomcat |
| | 1.000 | | App: ART | Apache Tomcat/6.0.18 on lc55-64-tomcat |
| | 1.000 | lc55-64-tomcat | Process: ART Front-end | Tomcat |
| | 0.732 | lc55-64-tomcat | Process Port: ART | 8780 on Tomcat on lc55-64-tomcat |
| | 0.729 | | App: Max RT | Apache Tomcat/6.0.18 on lc55-64-tomcat |
| | 0.729 | lc55-64-tomcat | Machine: Max RT Front-end | lc55-64-tomcat |
| | 0.729 | lc55-64-tomcat | Process: Max RT Front-end | Tomcat |
| | 0.693 | lc55-64-tomcat | Machine: ART Front-end | lc55-64-tomcat |
| | 0.664 | lc55-64-tomcat | Process Port: Max RT | 8780 on Tomcat on lc55-64-tomcat |
| | 0.557 | w2k8-64-oracle1 | Link: Network I/O to Back-end | Oracle App Server 1744 on w2k8-64-oracle1 to 192.168.2.10:1521 (Oracle TNS Listener) on w2k8-64-o... |
| | 0.538 | js22-2 | Disk: Total Read Wait Time | hdisk0 |
| | 0.535 | sol0u6-ld64-oracle | Process: CPU | nfsmapid |
| | 0.535 | js22-2 | Disk: Read I/O | hdisk0 |
| | 0.535 | js22-2 | Disk: Number of Reads | hdisk0 |
| | 0.535 | js22-2 | Process: Pages In | IBM(R) DB2(R) [js22-2-db297] 241866 |
| | 0.525 | lc55-64-tomcat | Machine: ART Back-end | lc55-64-tomcat |
| | 0.525 | lc55-64-tomcat | Process: ART Back-end | Tomcat |
| | 0.522 | lc55-64-tomcat | Process Port: TRT | 8780 on Tomcat on lc55-64-tomcat |
| | 0.521 | lc55-64-tomcat | Link: ART | Tomcat on lc55-64-tomcat to 3306 (mysqld) on lc55-32-databases |
| | 0.517 | js22-2 | Process: CPU | IBM(R) DB2(R) [js22-2-db297] 241866 |
| | 0.499 | w2k8-64-oracle1 | Link: ART | EMAgent on w2k8-64-oracle1 to 1158 (Oracle App Server 1744) on w2k8-64-oracle1 |
| | 0.499 | w2k8-64-oracle1 | Link: TRT | EMAgent on w2k8-64-oracle1 to 1158 (Oracle App Server 1744) on w2k8-64-oracle1 |
| | 0.492 | js22-2 | Process: CPU | snmpmibd |
| | 0.491 | lc55-32-databases | Process: ART Front-end | mysqld |

FIG. 18



Correlating ART for lc55-64-tomcat

Correlating ART for lc55-64-tomcat

Context: App Explorer: Apache Tomcat/6.0.18 on lc55-64-tomcat

Select Scope:
○ Direct dependencies only
● All metrics on all dependent machines

Filter Metrics:
☑ Transaction
□ Link
□ Process
□ Server Stack

Filter Results: GEO_ZONE_NAME (9 of 3501 entries)

Correlated Metric:
ART (Avg Response Time in ms)

| Correlated Data | R | Machine | Type | Metric Source |
|---|---|---|---|---|
| | 1.000 | lc55-64-tomcat | Machine: ART | lc55-64-tomcat |
| | 0.770 | lc55-64-tomcat | Tx Load | SQL Requests sent from Tomcat on lc55-64-tomcat to 3306 (mysqld) on lc55-32-databases matching ... |
| | 0.770 | lc55-64-tomcat | Tx Size | SQL Requests sent from Tomcat on lc55-64-tomcat to 3306 (mysqld) on lc55-32-databases matching ... |
| | 0.185 | w2k8-64-oracle1 | Tx ART | SQL Requests served by 1521 on Oracle TNS Listener on w2k8-64-oracle1 matching SELECT GEO_ZO... |
| | 0.135 | w2k8-64-oracle1 | Tx Load | SQL Requests served by 1521 on Oracle TNS Listener on w2k8-64-oracle1 matching SELECT GEO_ZO... |
| | 0.135 | w2k8-64-oracle1 | Tx Size | SQL Requests served by 1521 on Oracle TNS Listener on w2k8-64-oracle1 matching SELECT GEO_ZO... |
| | 0.124 | w2k8-64-oracle1 | Tx TRT | SQL Requests served by 1521 on Oracle TNS Listener on w2k8-64-oracle1 matching SELECT GEO_ZO... |
| | 0.124 | w2k8-64-oracle1 | Tx Max RT | SQL Requests served by 1521 on Oracle TNS Listener on w2k8-64-oracle1 matching SELECT GEO_ZO... |
| | 0.054 | w2k8-64-oracle1 | Tx Load | SQL Requests served by 1521 on Oracle TNS Listener on w2k8-64-oracle1 matching SELECT GEO_ZO... |
| | 0.054 | w2k8-64-oracle1 | Tx Size | SQL Requests served by 1521 on Oracle TNS Listener on w2k8-64-oracle1 matching SELECT GEO_ZO... |

FIG. 19

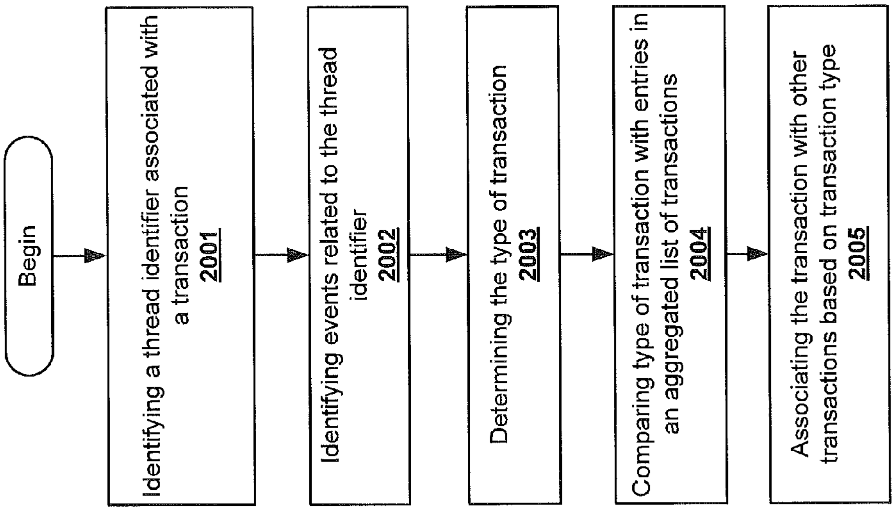

Begin
Identifying a thread identifier associated with a transaction
2001
Identifying events related to the thread identifier
2002
Determining the type of transaction
2003
Comparing type of transaction with entries in an aggregated list of transactions
2004
Associating the transaction with other transactions based on transaction type
2005

FIG. 20

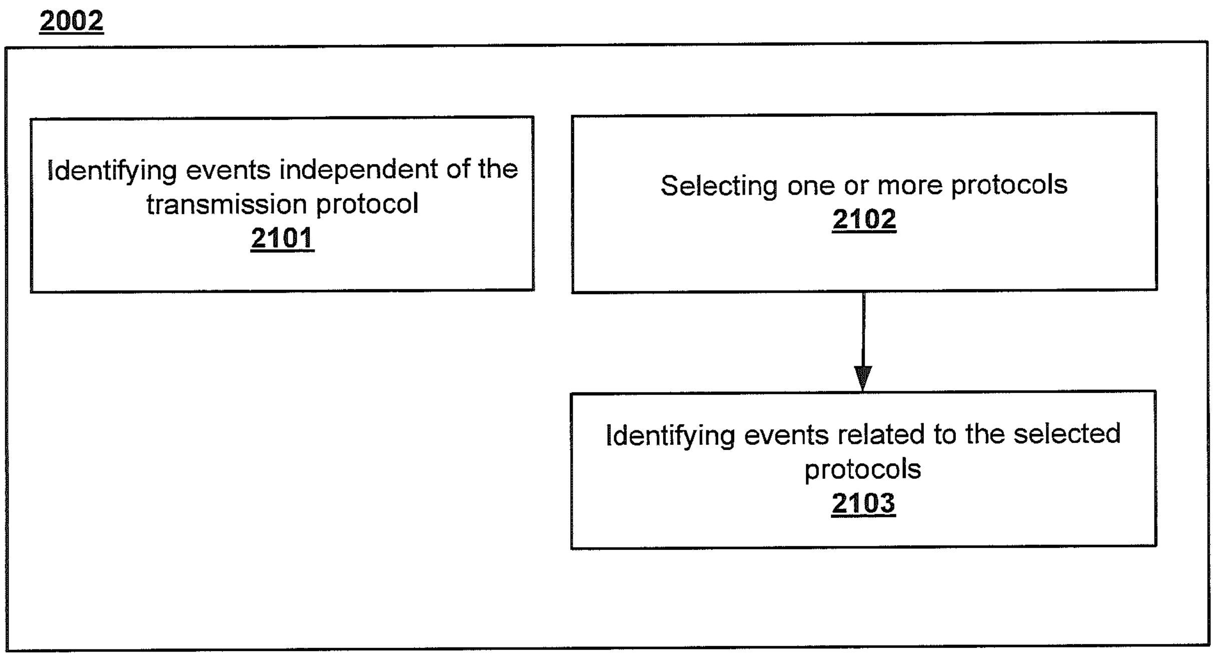

2002
Identifying events independent of the transmission protocol
2101
Selecting one or more protocols
2102
Identifying events related to the selected protocols
2103

FIG. 21

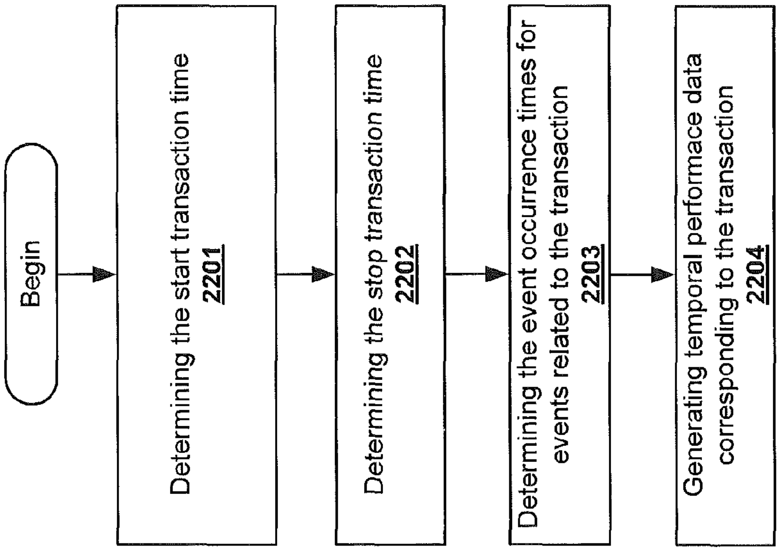

Begin
Determining the start transaction time
2201
Determining the stop transaction time
2202
Determining the event occurrence times for events related to the transaction
2203
Generating temporal performace data corresponding to the transaction
2204

FIG. 22

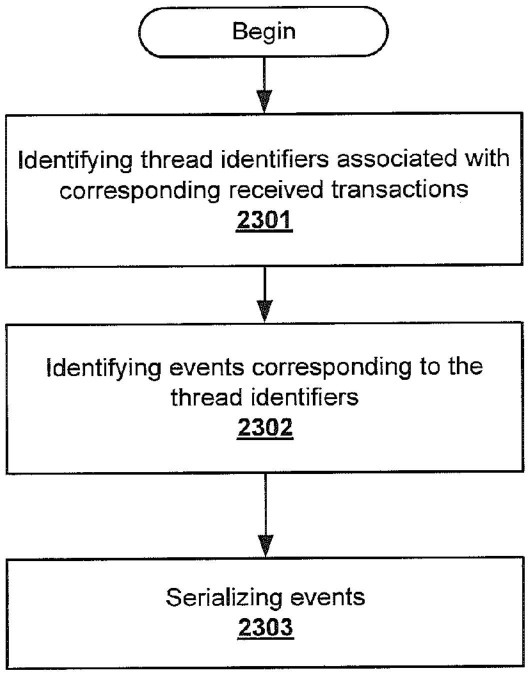

Begin
Identifying thread identifiers associated with corresponding received transactions
2301
Identifying events corresponding to the thread identifiers
2302
Serializing events
2303

FIG. 23

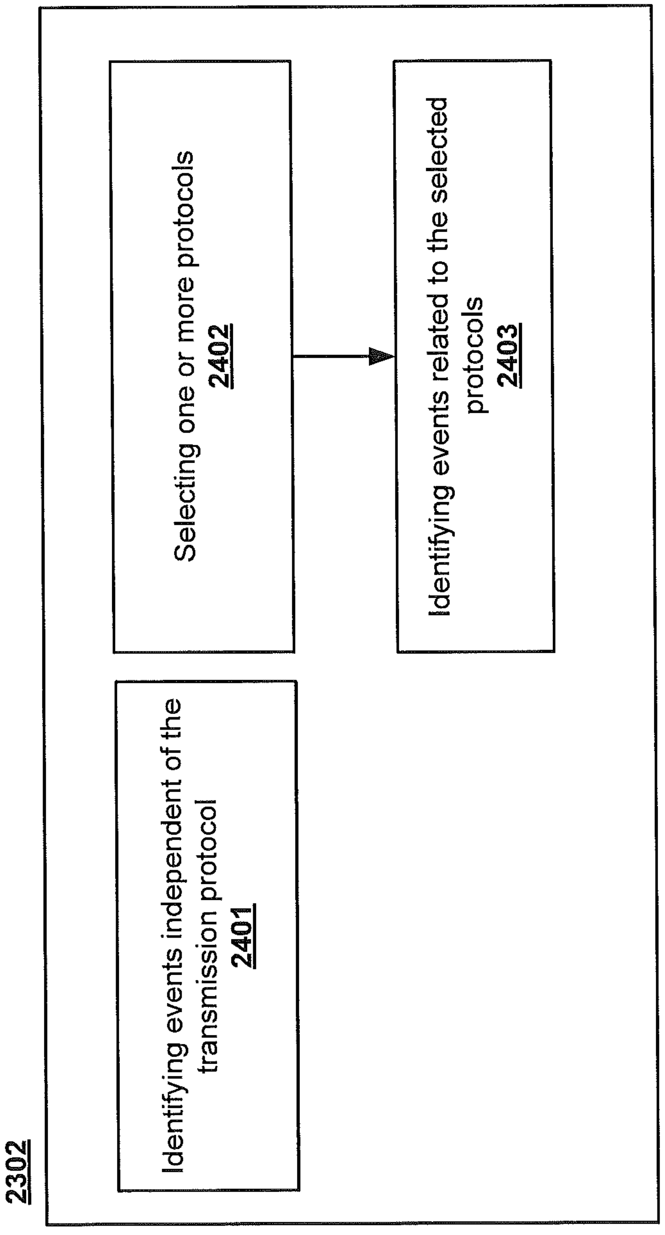

2302
Identifying events independent of the transmission protocol
2401
Selecting one or more protocols
2402
Identifying events related to the selected protocols
2403

FIG. 24

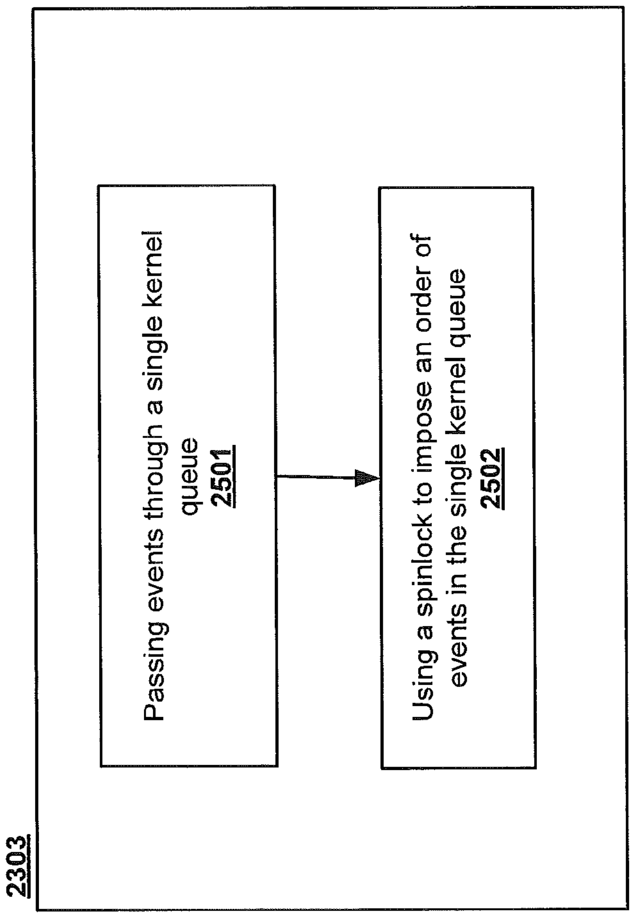

2303
Passing events through a single kernel queue
2501
Using a spinlock to impose an order of events in the single kernel queue
2502

FIG. 25

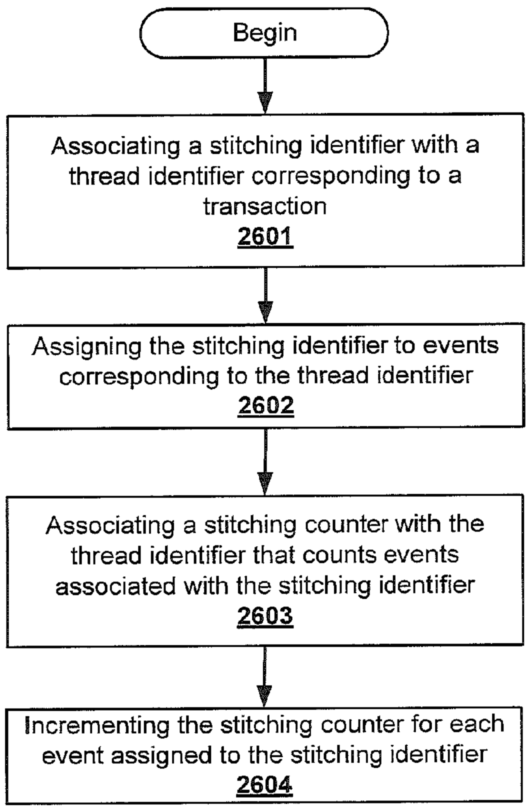

Begin
Associating a stitching identifier with a thread identifier corresponding to a transaction
2601
Assigning the stitching identifier to events corresponding to the thread identifier
2602
Associating a stitching counter with the thread identifier that counts events associated with the stitching identifier
2603
Incrementing the stitching counter for each event assigned to the stitching identifier
2604

FIG. 26

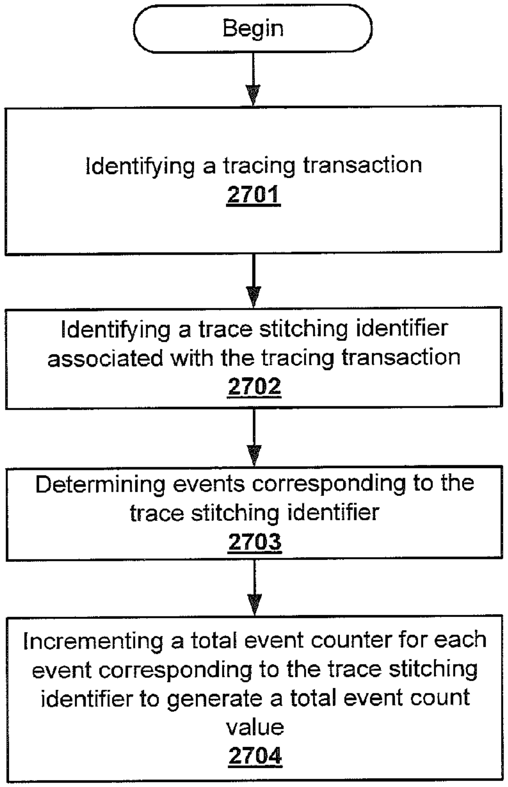

Begin
Identifying a tracing transaction
2701
Identifying a trace stitching identifier associated with the tracing transaction
2702
Determining events corresponding to the trace stitching identifier
2703
Incrementing a total event counter for each event corresponding to the trace stitching identifier to generate a total event count value
2704

FIG. 27

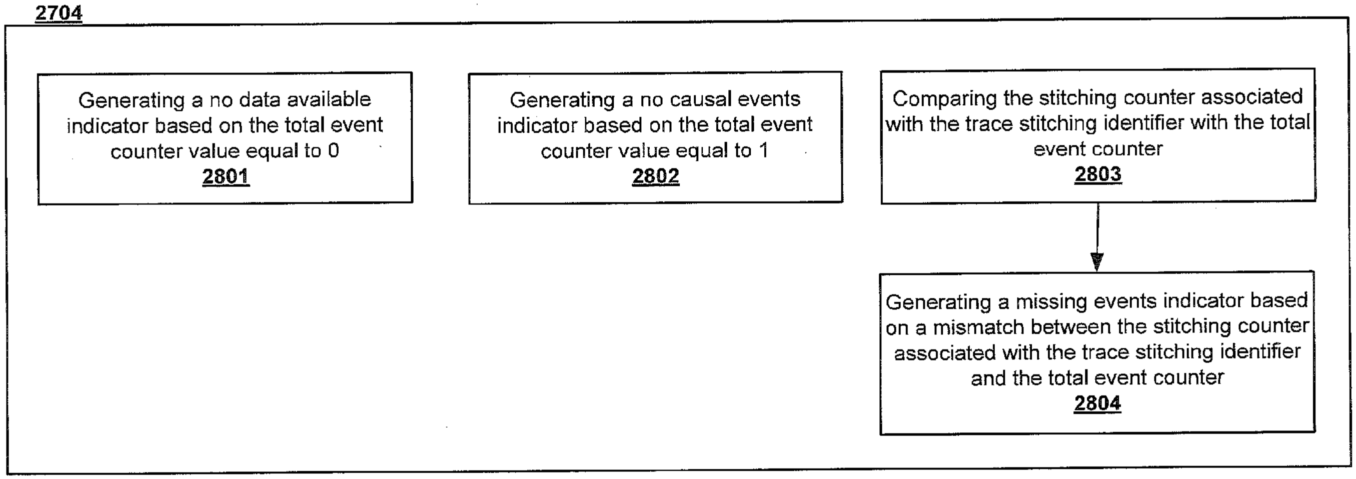

2704
Generating a no data available indicator based on the total event counter value equal to 0
2801
Generating a no causal events indicator based on the total event counter value equal to 1
2802
Comparing the stitching counter associated with the trace stitching identifier with the total event counter
2803
Generating a missing events indicator based on a mismatch between the stitching counter associated with the trace stitching identifier and the total event counter
2804

FIG. 28

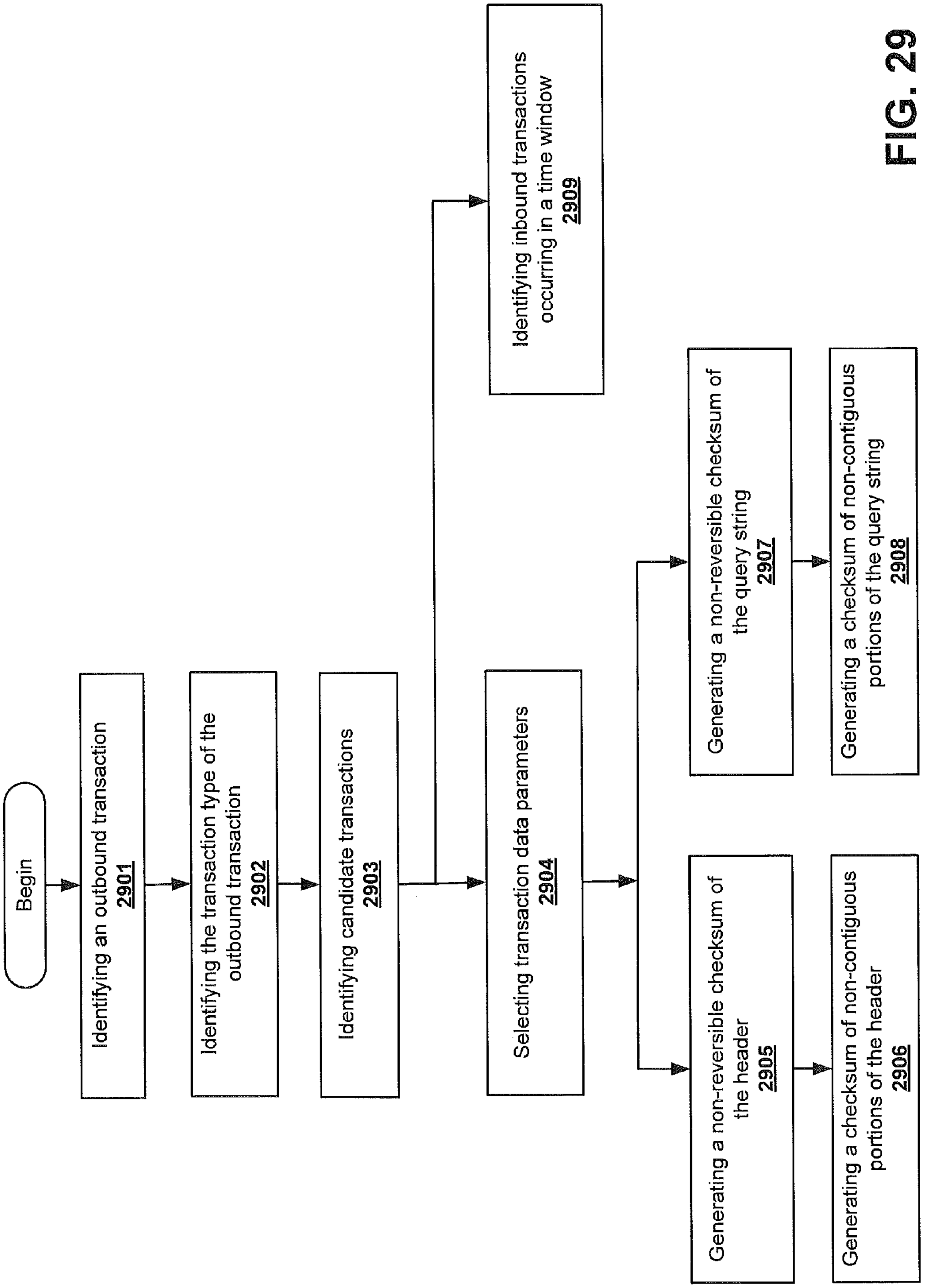

Begin
Identifying an outbound transaction 2901
Identifying the transaction type of the outbound transaction 2902
Identifying candidate transactions 2903
Selecting transaction data parameters 2904
Generating a non-reversible checksum of the header 2905
Generating a checksum of non-contiguous portions of the header 2906
Generating a non-reversible checksum of the query string 2907
Generating a checksum of non-contiguous portions of the query string 2908
Identifying inbound transactions occurring in a time window 2909

FIG. 29

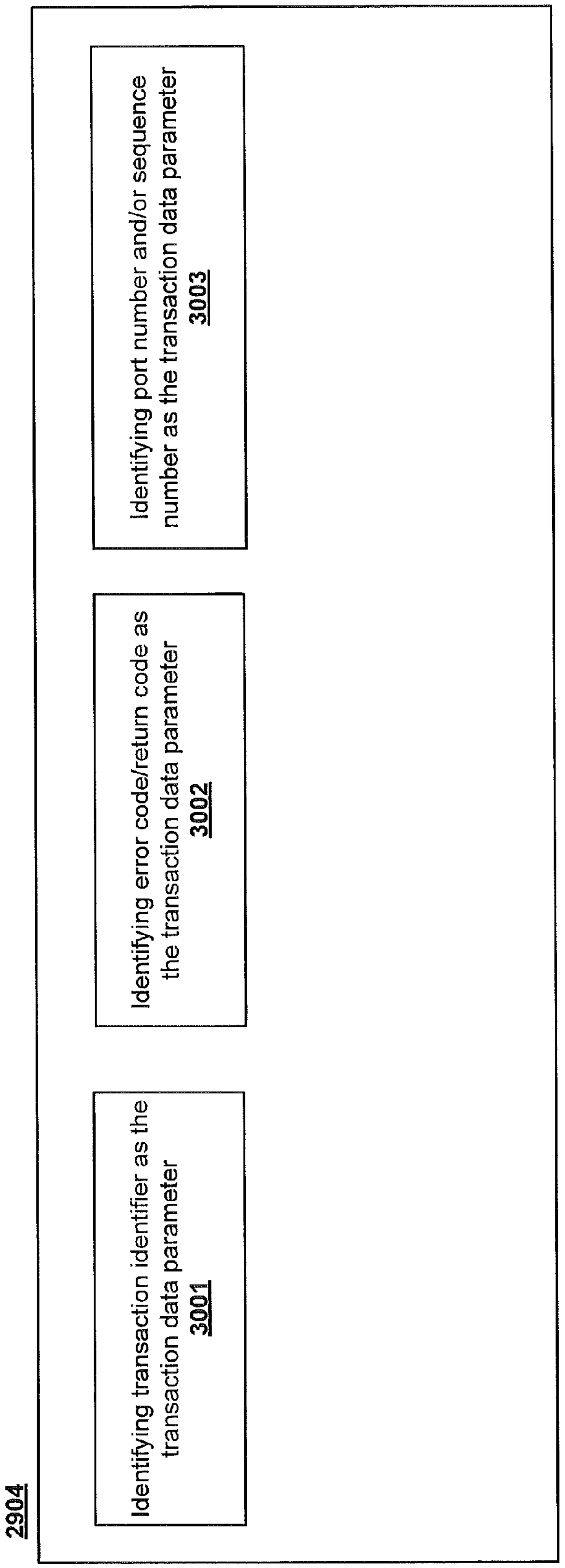

2904
Identifying transaction identifier as the transaction data parameter
3001
Identifying error code/return code as the transaction data parameter
3002
Identifying port number and/or sequence number as the transaction data parameter
3003

FIG. 30

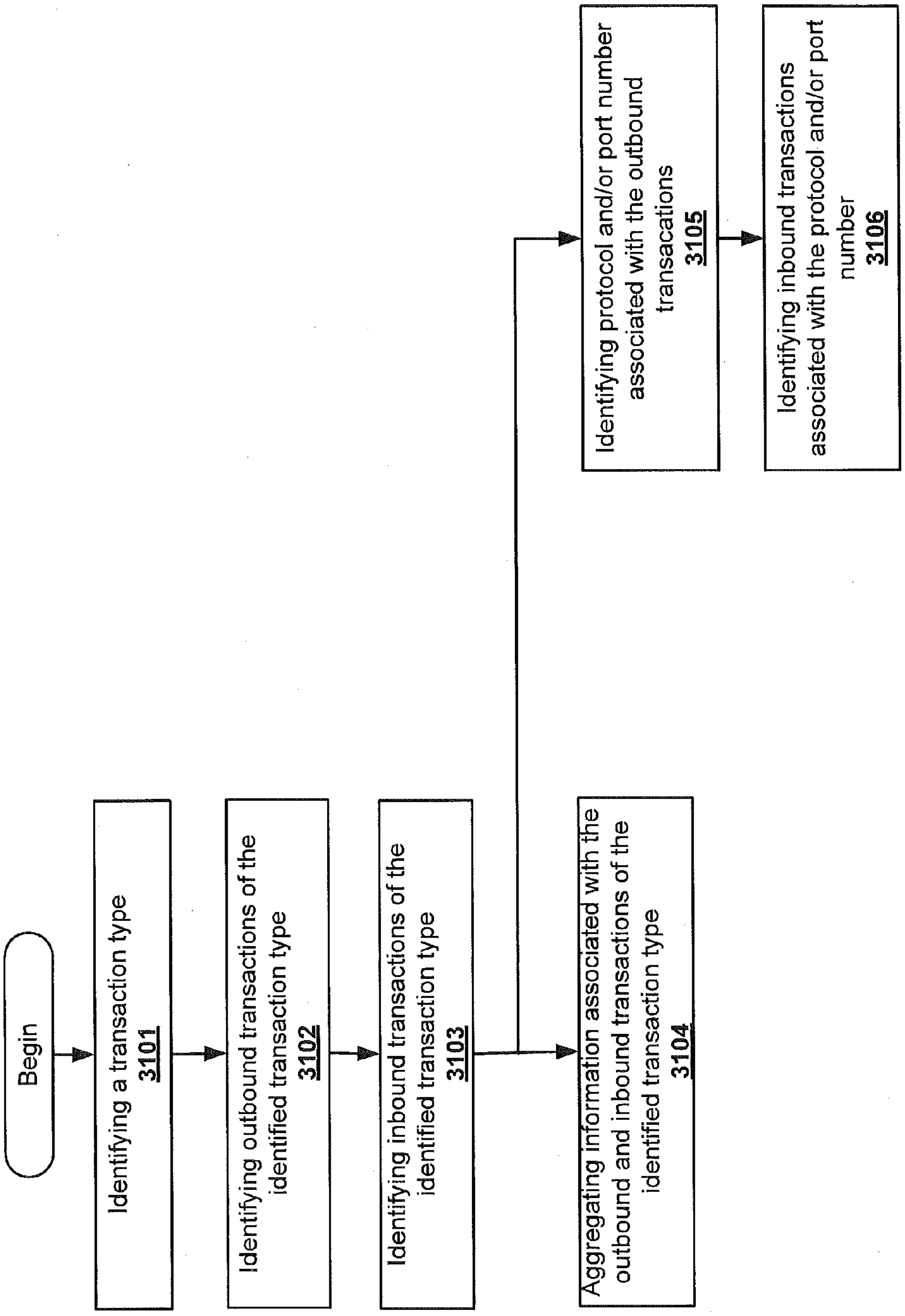
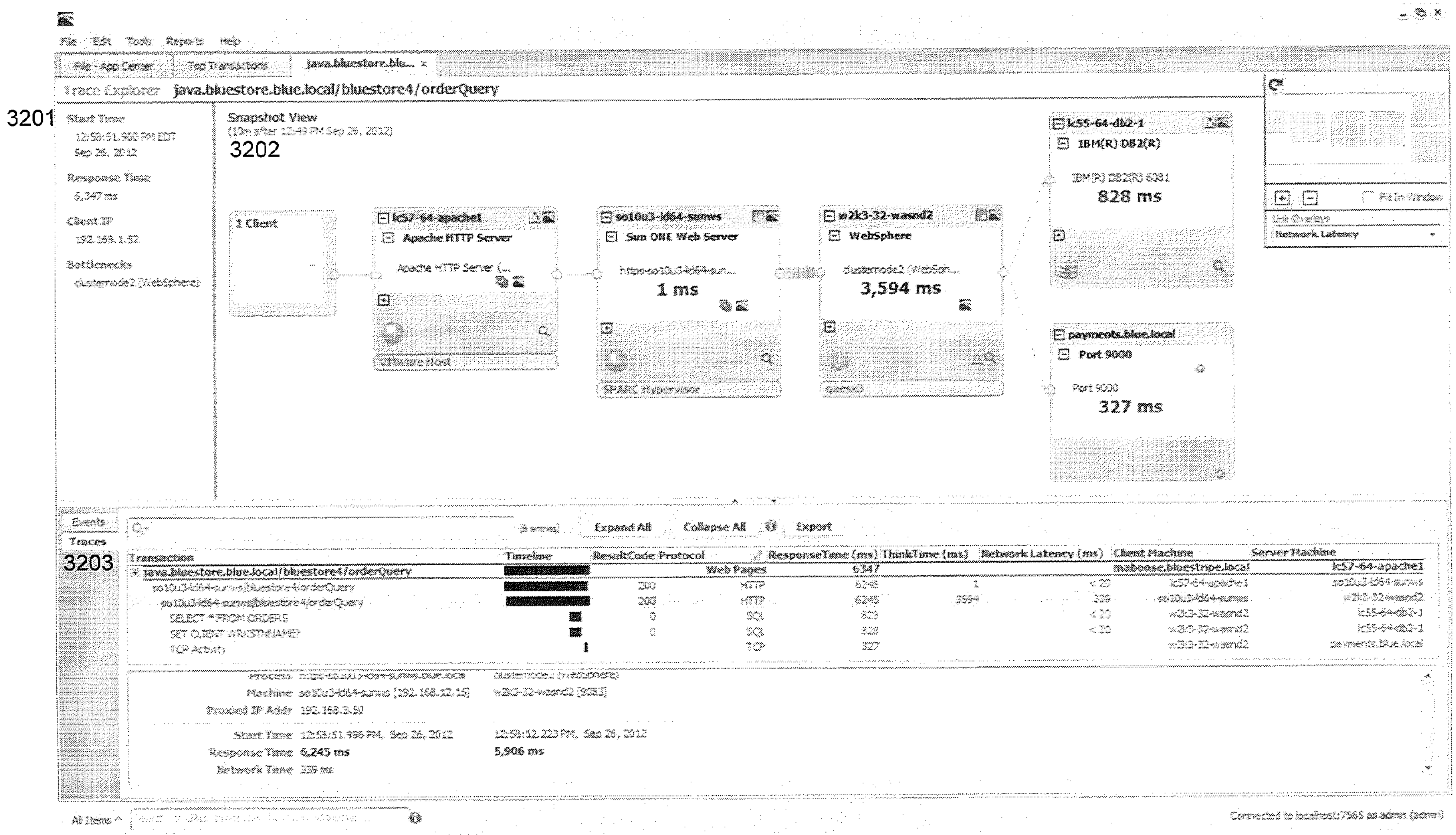

Begin
Identifying a transaction type
3101
Identifying outbound transactions of the identified transaction type
3102
Identifying inbound transactions of the identified transaction type
3103
Aggregating information associated with the outbound and inbound transactions of the identified transaction type
3104
Identifying protocol and/or port number associated with the outbound transacations
3105
Identifying inbound transactions associated with the protocol and/or port number
3106

FIG. 31



Start Time
12:58:51.900 PM EDT
Sep 26, 2012

Response Time
6,347 ms

Client IP
192.168.1.52

Bottlenecks
clusternode2 (WebSphere)

| Transaction | Timeline | ResultCode | Protocol | ResponseTime (ms) | ThinkTime (ms) | Network Latency (ms) | Client Machine | Server Machine |
|---|---|---|---|---|---|---|---|---|
| java.bluestore.blue.local/bluestore4/orderQuery | ▇▇▇▇ | | Web Pages | 6347 | | | maboose.bluestripe.local | lc57-64-apache1 |
| so10u3-ld64-sunws/bluestore4/orderQuery | ▇▇▇▇ | 200 | HTTP | 6248 | 1 | < 20 | lc57-64-apache1 | so10u3-ld64-sunws |
| so10u3-ld64-sunws/bluestore4/orderQuery | ▇▇▇▇ | 200 | HTTP | 6245 | 3594 | 339 | so10u3-ld64-sunws | w2k3-32-wasnd2 |
| SELECT * FROM ORDERS | ▇ | 0 | SQL | 829 | | < 20 | w2k3-32-wasnd2 | lc55-64-db2-1 |
| SET CLIENT WRKSTNNAME? | ▇ | 0 | SQL | 828 | | < 20 | w2k3-32-wasnd2 | lc55-64-db2-1 |
| TCP Activity | ▏ | | TCP | 327 | | | w2k3-32-wasnd2 | payments.blue.local |

| | | |
|---|---|---|
| Process | https-so10u3-ld64-sunws.blue.local | clusternode2 (WebSphere) |
| Machine | so10u3-ld64-sunws [192.168.22.15] | w2k3-32-wasnd2 [9083] |
| Proxied IP Addr | 192.168.3.91 | |
| Start Time | 12:58:51.996 PM, Sep 26, 2012 | 12:58:52.223 PM, Sep 26, 2012 |
| Response Time | 6,245 ms | 5,906 ms |
| Network Time | 339 ms | |

FIG. 32

# METHODS AND COMPUTER PROGRAM PRODUCTS FOR TRANSACTION ANALYSIS OF NETWORK TRAFFIC IN A NETWORK DEVICE

## FIELD OF INVENTION

The present invention relates to computer networks and, more particularly, to network performance monitoring methods, devices, and computer program products.

## BACKGROUND

The growing presence of computer networks such as intranets and extranets has brought about the development of applications in e-commerce, education, manufacturing, and other areas. Organizations increasingly rely on such applications to carry out their business, production, or other objectives, and devote considerable resources to ensuring that the applications perform as expected. To this end, various application management, monitoring, and analysis techniques have been developed.

One approach for managing an application involves monitoring the application, generating data regarding application performance, and analyzing the data to determine application health. Some system management products analyze a large number of data streams to try to determine a normal and abnormal application state. Large numbers of data streams are often analyzed because the system management products may not have a semantic understanding of the data being analyzed. Accordingly, when an unhealthy application state occurs, many data streams may have abnormal data values because the data streams are causally related to one another. Because the system management products may lack a semantic understanding of the data, they may not be able to assist the user in determining either the ultimate source or cause of a problem. Additionally, these application management systems may not know whether a change in data indicates an application is actually unhealthy or not.

Current application management approaches may include monitoring techniques such as deep packet inspection (DPI), which may be performed as a packet passes an inspection point and may include collecting statistical information, among others. Such monitoring techniques can be data-intensive and may be ineffective in providing substantively real-time health information regarding network applications. Additionally, packet trace information may be lost and application-specific code may be required.

Embodiments of the present invention are, therefore, directed towards solving these and other related problems.

## SUMMARY

It should be appreciated that this Summary is provided to introduce a selection of concepts in a simplified form, the concepts being further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of this disclosure, nor is it intended to limit the scope of the invention.

Some embodiments of the present invention are directed to methods for analyzing network traffic transactions. Methods may include identifying a thread identifier associated with a transaction and identifying one or more events related to the thread identifier. The type of the transaction may be determined to be a first type of transaction. Some embodiments may include comparing the first type of transaction with respective types of transactions among entries in an aggregated list of transactions. The transaction and the one or more events related to the thread identifier associated with the transaction may be associated with an entry in the aggregated list of transactions based on a match between the first type of transaction and a respective one of the types of transactions associated with the entry in the aggregated list of transactions. The aggregated list of transactions may include one or more transactions received by an application server in a time window.

In some embodiments, a start transaction time associated with a start of the transaction and a stop transaction time associated with completion of the transaction may be determined. Some embodiments may include determining an event time for respective ones of the one or more events related to the thread identifier associated with the transaction. The start of the transaction may correspond to a receipt of a request for the transaction. In some embodiments, a difference between the stop transaction time and the start transaction time may be compared to a threshold. Temporal performance data corresponding to the transaction based on the start transaction time, the stop transaction time, and/or the event times may be generated. The temporal performance data may include statistical data.

In some embodiments the methods described herein may be applied to a web server identified to use thread identifiers as valid join identifiers. The web server may include an Apache server, a Sun ONE server and/or a Java-based application server. The thread identifier may be unique among threads across the web server.

Identifying one or more events, according to some embodiments, may include identifying one or more events that use any of the one or more networking protocols used by the respective transaction. Identifying one or more events may include selecting one or more protocols and identifying one or more events using the selected protocols. The transaction may also use a selected network protocol.

Some embodiments of the present invention are directed to a method of tracing transactions and related events of network traffic sent to and/or received by a network device. Methods may include identifying a plurality of thread identifiers, ones of which may be associated with corresponding ones of a plurality of received transactions. Some embodiments may include identifying one or more events corresponding to the respective ones of the plurality of thread identifiers. The transaction and one or more events related to the plurality of thread identifiers may be serialized into a list of events and/or an ordered list.

According to some embodiments, a stitching identifier may be associated with a transaction of the plurality of received transactions. The stitching identifier may be assigned to all events corresponding to any specific thread identifier. A stitching counter counts events associated with the stitching identifier. The stitching counter may be incremented for each of the events assigned to the stitching identifier.

In some embodiments a tracing transaction and the trace stitching identifier associated with the tracing transaction may be identified. Embodiments of the method may include determining, from the serialized list of events, one or more events corresponding to the trace stitching identifier to identify events that correspond to the tracing transaction. A total event counter may be incremented for events corresponding to the trace stitching identifier in order to generate a total event count value.

According to some embodiments, a no data available indicator may be generated based on the total event counter value equal to 0. A no causal events indicator may be generated based on the total event counter value equal to 1. In some embodiments, the stitching counter associated with the trace stitching identifier may be compared with the total event counter and a missing events indicator may be generated based on a mismatch between the stitching counter associated with the trace stitching identifier and the total event counter.

In some embodiments, the respective thread identifiers of the plurality of threads may be unique in an application server with respect to a kernel. Serializing the events may include passing events through a single kernel queue and/or using a spinlock that imposes an order of events in the single kernel queue. The thread identifiers may be represented by an address location in the kernel of a control block of the thread and/or the thread identifiers may be represented by a pointer to an address location in the kernel of a control block of the thread.

In some embodiments, identifying one or more events may include identifying one or more events that include any of the one or more networking protocols used by the corresponding received transaction. Identifying one or more events may include selecting one or more protocols and identifying one or more events associated with the selected protocols used by the corresponding received transaction. The selected protocols may be associated with one of the received transactions.

Some embodiments of the present invention may be directed to methods of identifying one or more events that are associated with a transaction based on a thread identifier and associating the transaction and the one or more events with at least one transaction of a plurality of transactions based on a transaction type.

In some embodiments, a single transaction may be recorded at two ends of a communication link. The same transaction may be recorded as an outbound transaction at a sender and as an inbound transaction at a receiver. Some embodiments of the present invention may identify which pairs of outbound and inbound messages correspond to single transactions. The transaction type of each outbound transaction may be identified. One or more inbound transactions which have the same transaction type as the transaction type of an outbound transaction may be identified as candidate transactions. One or more transaction data parameters may be selected that correspond to the outbound and/or inbound transactions. A candidate outbound transaction may be determined to be related to a candidate inbound transaction based on the transaction data parameters. The inbound transactions may include transactions received in a time window.

In some embodiments, a transaction identifier may be identified as one of the transaction data parameters. An error code and/or a return code may be identified as one of the transaction data parameters. Selecting one or more transaction data parameters may include identifying a port number and/or a sequence number as one of the transaction data parameters. The sequence number may be determined based on a byte offset of a TCP stream. The one or more transaction data parameters may include a protocol specific identifier. The protocol specific identifier may comprise a header and the header may be a Hypertext Transfer Protocol (HTTP) request header.

In some embodiments, a checksum of one or more contiguous or non-contiguous portions of at least a portion of the header may be generated. A non-reversible checksum of one or more contiguous or non-contiguous portions of at least a portion of the header may be generated.

In some embodiments, the protocol specific identifier may include a query string. A non-reversible checksum of at least a portion the query string may be generated. A checksum of one or more non-contiguous portions of at least a portion of the query string may be generated.

According to some embodiments, a transaction type for analysis of one or more application servers may be identified. A plurality of outbound transactions of the transaction type identified for analysis may be determined. A plurality of inbound transactions of the transaction type identified for analysis may be determined. Information associated with the plurality of outbound transactions and information associated with the plurality of inbound transactions of a same transaction type may be aggregated.

In some embodiments, a protocol and/or a port number associated with each of the plurality of outbound transactions from an application server may be identified. A plurality of related inbound transactions for one or more different web servers that are based on the protocol and/or the port number associated with the plurality of outbound transactions may be identified.

In some embodiments, a computer program product including a non-transitory computer usable storage medium having computer-readable program code embodied in the medium is provided. The computer-readable program code is configured to perform operations corresponding to methods described herein.

It is noted that aspects of the inventive concepts described with respect to one embodiment may be incorporated in a different embodiment although not specifically described relative thereto. That is, all embodiments and/or features of any embodiments can be combined in any way and/or combination. These and other objects and/or aspects of the present inventive concepts are explained in detail in the specification set forth below.

Other methods, devices, and/or computer program products according to example embodiments will be or become apparent to one with skill in the art upon review of the following drawings and detailed description. It is intended that all such additional methods, devices, and/or computer program products be included within this description, be within the scope of the present invention, and be protected by the accompanying claims.

## BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will now be described in more detail in relation to the enclosed drawings, in which:

FIGS. **1***a*-**1***d* are block diagrams illustrating example networks in which operations for monitoring network application performance may be performed according to some embodiments of the present invention.

FIG. **2** is a block diagram illustrating an architecture of a computing device as discussed above regarding FIGS. **1***c* and **1***d*.

FIG. **3** is a block diagram illustrating operations and/or functions of a collector application as described above regarding FIG. **1***a*.

FIG. **4** is a diagram illustrating determining a read wait time corresponding to a user transaction according to some embodiments of the present invention.

FIG. **5** is a block diagram illustrating a kernel level architecture of a collector application to explain kernel level metrics according to some embodiments of the present invention.

FIG. **6** is a flowchart illustrating example operations carried out by a collector application in monitoring and reporting network application performance according to some embodiments of the present invention.

FIG. **7** is a screen shot of a graphical user interface (GUI) including a model generated by a health data processing application according to some embodiments of the present invention.

FIG. **8** is a flowchart illustrating example operations carried out by a health data processing application in generating and displaying a real-time model of network application health according to some embodiments of the present invention.

FIG. **9** is a flowchart illustrating example operations carried out by a health data processing application in generating and displaying an historical model of network application health according to some embodiments of the present invention.

FIG. **10** is a block diagram illustrating the architecture of a system providing network traffic data parsing and filtering according to some embodiments of the present invention.

FIG. **11** is a flowchart illustrating example operations carried out by a collector application in collecting raw network traffic data that is sent to and/or received by a network device according to some embodiments of the present invention.

FIG. **12** is a flowchart illustrating example operations carried out by a collector application in parsing and filtering collected network traffic data according to some embodiments of the present invention.

FIG. **13** is a flowchart illustrating example operations carried out by a correlation analyzer for analyzing correlation of collected network traffic data metrics according to some embodiments of the present invention.

FIG. **14** is a flowchart illustrating example settings that may be applied to define the candidate set according to some embodiments of the present invention.

FIG. **15** is a flowchart illustrating example user input operations to control temporal shifting during correlation analysis according to some embodiments of the present invention.

FIG. **16** is a flowchart illustrating example operations in methods/systems that generate the network metric type for use in identifying the candidate set for correlation analysis according to some embodiments of the present invention.

FIG. **17** is a flowchart illustrating example user input operations to control temporal shifting during correlation analysis according to some embodiments of the present invention.

FIG. **18** is a screen shot of a graphical user interface (GUI) including selection options for scope and network metric types for correlation analysis and display of results related to a primary metric by a correlation analysis application according to some embodiments of the present invention.

FIG. **19** is a screen shot of a graphical user interface (GUI) including an example selection of transaction as the network metric type for correlation analysis and display of results related to a primary metric by a correlation analysis application according to some embodiments of the present invention.

FIG. **20** is a flowchart illustrating example operations carried out by a health data processing application to generate an aggregated list of transactions based on the thread identifier and transaction type according to some embodiments of the present invention.

FIG. **21** is a flowchart illustrating example operations carried out by a health data processing application to identify transaction events based on networking protocols according to some embodiments of the present invention.

FIG. **22** is a flowchart illustrating example operations carried out by a health data processing application to generate temporal performance data based on transaction and event times according to some embodiments of the present invention.

FIG. **23** is a flowchart illustrating example operations carried out by a health data processing application to identify and serialize events based on the thread identifiers of transactions according to some embodiments of the present invention.

FIG. **24** is a flowchart illustrating example operations carried out by a health data processing application to identify transaction events based on networking protocols according to some embodiments of the present invention.

FIG. **25** is a flowchart illustrating example operations carried out by a health data processing application to serialize events by passing events through a single kernel queue and/or using a spinlock according to some embodiments of the present invention.

FIG. **26** is a flowchart illustrating example operations carried out by a health data processing application to assign a stitching identifier and maintain a stitching counter for events based on the thread identifier according to some embodiments of the present invention.

FIG. **27** is a flowchart illustrating example operations carried out by a health data processing application to determine and count events for a transaction to be traced according to some embodiments of the present invention.

FIG. **28** is a flowchart illustrating example operations carried out by a health data processing application to generate indicators based on front-to-back stitching according to some embodiments of the present invention.

FIG. **29** is a flowchart illustrating example operations carried out by a health data processing application in trace mode based on back-to-front stitching according to some embodiments of the present invention.

FIG. **30** is a flowchart illustrating example operations carried out by a health data processing application for selecting transaction data parameters in trace mode using back-to-front stitching according to some embodiments of the present invention.

FIG. **31** is a flowchart illustrating example operations carried out by a health data processing application in an aggregate mode using back-to-front stitching according to some embodiments of the present invention.

FIG. **32** is a screen shot of a graphical user interface (GUI) including example tracing of transaction events according to some embodiments of the present invention.

## DETAILED DESCRIPTION

In the following description, for purposes of explanation and not limitation, specific details are set forth such as particular architectures, interfaces, techniques, etc. in order to provide a thorough understanding of the present invention. However, it will be apparent to those skilled in the art that the present invention may be practiced in other embodiments that depart from these specific details. In other instances, detailed descriptions of well known devices, circuits, and methods are omitted so as not to obscure the description of the present invention with unnecessary detail. While various modifications and alternative forms of the embodiments described herein may be made, specific embodiments are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that there is no intent to limit the invention to the particular forms disclosed, but on the contrary, the invention is to cover all modifications, equivalents, and alternatives falling within the spirit and scope of the invention as defined by the claims. Like reference numbers signify like elements throughout the description of the figures.

As used herein, the singular forms "a," "an," and "the" are intended to include the plural forms as well, unless expressly stated otherwise. It should be further understood that the terms "comprises" and/or "comprising" when used in this specification are taken to specify the presence of stated features, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, steps, operations, elements, components, and/or groups thereof. It will be understood that when an element is referred to as being "connected" or "coupled" to another element, it can be directly connected or coupled to the other element or intervening elements may be present. Furthermore, "connected" or "coupled" as used herein may include wirelessly connected or coupled. As used herein, the term "and/or" includes any and all combinations of one or more of the associated listed items, and may be abbreviated as "/".

Unless otherwise defined, all terms (including technical and scientific terms) used herein have the same meaning as commonly understood by one of ordinary skill in the art. It will be further understood that terms, such as those defined in commonly used dictionaries, should be interpreted as having a meaning that is consistent with their meaning in the context of the relevant art, and will not be interpreted in an idealized or overly formal sense unless expressly so defined herein.

It will be understood that, although the terms first, second, etc, may be used herein to describe various elements, these elements should not be limited by these terms. These terms are only used to distinguish one element from another.

Example embodiments are described below with reference to block diagrams and/or flowchart illustrations of methods, apparatus (systems and/or devices), and/or computer program products. It is understood that a block of the block diagrams and/or flowchart illustrations, and combinations of blocks in the block diagrams and/or flowchart illustrations, can be implemented by computer program instructions. These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, and/or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer and/or other programmable data processing apparatus, create means (functionality) and/or structure for implementing the functions/acts specified in the block diagrams and/or flowchart block or blocks.

These computer program instructions may also be stored in a computer-readable memory that can direct a computer or other programmable data processing apparatus to function in a particular manner, such that the instructions stored in the computer-readable memory produce an article of manufacture including instructions which implement the functions/acts specified in the block diagrams and/or flowchart block or blocks.

The computer program instructions may also be loaded onto a computer or other programmable data processing apparatus to cause a series of operational steps to be performed on the computer or other programmable apparatus to produce a computer-implemented process, such that the instructions, which execute on the computer or other programmable apparatus, provide steps for implementing the functions/acts specified in the block diagrams and/or flowchart block or blocks.

Accordingly, example embodiments may be implemented in hardware and/or in software (including firmware, resident software, micro-code, etc.). Furthermore, example embodiments may take the form of a computer program product on a non-transitory computer-usable or computer-readable storage medium having computer-usable or computer-readable program code embodied in the medium for use by or in connection with an instruction execution system. In the context of this document, a non-transitory computer-usable or computer-readable medium may be any medium that can contain, store, or transport the program for use by or in connection with the instruction execution system, apparatus, or device.

The computer-usable or computer-readable medium may be, for example but not limited to, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, apparatus, or device. More specific examples (a non-exhaustive list) of the computer-readable medium would include the following: a portable computer diskette, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), and a portable compact disc read-only memory (CD-ROM).

Computer program code for carrying out operations of data processing systems discussed herein may be written in a high-level programming language, such as C, C++, or Java, for development convenience. In addition, computer program code for carrying out operations of example embodiments may also be written in other programming languages, such as, but not limited to, interpreted languages. Some modules or routines may be written in assembly language or even micro-code to enhance performance and/or memory usage. However, embodiments are not limited to a particular programming language. It will be further appreciated that the functionality of any or all of the program modules may also be implemented using discrete hardware components, one or more application specific integrated circuits (ASICs), or a programmed digital signal processor or microcontroller.

It should also be noted that in some alternate implementations, the functions/acts noted in the blocks may occur out of the order noted in the flowcharts. For example, two blocks shown in succession may in fact be executed substantially concurrently or the blocks may sometimes be executed in the reverse order, depending upon the functionality/acts involved. Moreover, the functionality of a given block of the flowcharts and/or block diagrams may be separated into multiple blocks and/or the functionality of two or more blocks of the flowcharts and/or block diagrams may be at least partially integrated.

Reference is made to FIGS. **1***a*-**1***d*, which are block diagrams illustrating example networks in which operations for monitoring and reporting network application performance may be performed according to some embodiments of the present invention.

Computing Network

Referring to FIG. **1***a*, a network **10** according to some embodiments herein may include a health data processing application **100** and a plurality of network devices **20**, **24**, and **26** that may each include respective collector applications **200**. It is to be understood that a "network device" as discussed herein may include physical (as opposed to virtual) machines **20**; host machines **24**, each of which may be a physical machine on which one or more virtual machines may execute; and/or virtual machines **26** executing on host machines **24**. It is to be further understood that an "application" as discussed herein refers to an instance of executable software operable to execute on respective ones of the network devices. The terms "application" and "network application" may be used interchangeably herein, regardless of whether the referenced application is operable to access network resources.

Collector applications **200** may collect data related to the performance of network applications executing on respective network devices. For instance, a collector application executing on a physical machine may collect performance data related to network applications executing on that physical machine. A collector application executing on a host machine and external to any virtual machines hosted by that host machine may collect performance data related to network applications executing on that host machine, while a collector application executing on a virtual machine may collect performance data related to network applications executing within that virtual machine.

The health data processing application **100** may be on a network device that exists within the network **10** or on an external device that is coupled to the network **10**. Accordingly, in some embodiments, the network device on which the health data processing application **100** may reside may be one of the plurality of machines **20** or **24** or virtual machines **26**. Communications between various ones of the network devices may be accomplished using one or more communications and/or network protocols that may provide a set of standard rules for data representation, signaling, authentication and/or error detection that may be used to send information over communications channels therebetween. In some embodiments, example network protocols may include Hypertext Transfer Protocol (HTTP), Tabular Data Stream (TDS), and/or Lightweight Directory Access Protocol (LDAP), among others.

Referring to FIG. **1***b*, an example network **10** may include a web server **12**, one or more application servers **14** and one or more database servers **16**. Although not illustrated, a network **10** as used herein may include directory servers, security servers, and/or transaction monitors, among others. The web server **12** may be a computer and/or a computer program that is responsible for accepting HTTP requests from clients **18** (e.g., user agents such as web browsers) and serving them HTTP responses along with optional data content, which may be, for example, web pages such as HTML documents and linked objects (images, etc.). An application server **14** may include a service, hardware, and/or software framework that may be operable to provide one or more programming applications to clients in a network. Application servers **14** may be coupled to one or more web servers **12**, database servers **16**, and/or other application servers **14**, among others. Some embodiments provide that a database server **16** may include a computer and/or a computer program that provides database services to other computer programs and/or computers as may be defined, for example by a client-server model, among others. In some embodiments, database management systems may provide database server functionality.

Some embodiments provide that the collector applications **200** and the health data processing application **100** described above with respect to FIG. **1***a* may reside on ones of the web server(s) **12**, application servers **14** and/or database servers **16**, among others. In some embodiments, the health data processing application **100** may reside in a dedicated computing device that is coupled to the network **10**. The collector applications **200** may reside on one, some or all of the above listed network devices and provide network application performance data to the health data processing application **100**.

Computing Device

Web server(s) **12**, application servers **14** and/or database servers **16** may be deployed as and/or executed on any type and form of computing device, such as a computer, network device, or appliance capable of communicating on any type and form of network and performing the operations described herein. FIGS. **1***c* and **1***d* depict block diagrams of a computing device **121** useful for practicing some embodiments described herein. Referring to FIGS. **1***c* and **1***d*, a computing device **121** may include a central processing unit **101** and a main memory unit **122**. A computing device **121** may include a visual display device **124**, a keyboard **126**, and/or a pointing device **127**, such as a mouse. Each computing device **121** may also include additional optional elements, such as one or more input/output devices **130***a*-**130***b* (generally referred to using reference numeral **130**), and a cache memory **140** in communication with the central processing unit **101**.

The central processing unit **101** is any logic circuitry that responds to and processes instructions fetched from the main memory unit **122**. In many embodiments, the central processing unit **101** is provided by a microprocessor unit, such as: those manufactured by Intel Corporation of Mountain View, Calif.; those manufactured by Motorola Corporation of Schaumburg, Ill.; the POWER processor, those manufactured by International Business Machines of White Plains, N.Y.; and/or those manufactured by Advanced Micro Devices of Sunnyvale, Calif. The computing device **121** may be based on any of these processors, and/or any other processor capable of operating as described herein.

Main memory unit **122** may be one or more memory chips capable of storing data and allowing any storage location to be directly accessed by the microprocessor **101**, such as Static random access memory (SRAM), Burst SRAM or SynchBurst SRAM (BSRAM), Dynamic random access memory (DRAM), Fast Page Mode DRAM (FPM DRAM), Enhanced DRAM (EDRAM), Extended Data Output RAM (EDO RAM), Extended Data Output DRAM (EDO DRAM), Burst Extended Data Output DRAM (BEDO DRAM), Enhanced DRAM (EDRAM), synchronous DRAM (SDRAM), JEDEC SRAM, PC100 SDRAM, Double Data Rate SDRAM (DDR SDRAM), Enhanced SDRAM (ESDRAM), SyncLink DRAM (SLDRAM), Direct Rambus DRAM (DRDRAM), or Ferroelectric RAM (FRAM), among others. The main memory **122** may be based on any of the above described memory chips, or any other available memory chips capable of operating as described herein. In some embodiments, the processor **101** communicates with main memory **122** via a system bus **150** (described in more detail below). In some embodiments of a computing device **121**, the processor **101** may communicate directly with main memory **122** via a memory port **103**. Some embodiments provide that the main memory **122** may be DRDRAM.

FIG. **1***d* depicts some embodiments in which the main processor **101** communicates directly with cache memory **140** via a secondary bus, sometimes referred to as a backside bus. In some other embodiments, the main processor **101** may communicate with cache memory **140** using the system bus **150**. Cache memory **140** typically has a faster response time than main memory **122** and may be typically provided by SRAM, BSRAM, or EDRAM. In some embodiments, the processor **101** communicates with various I/O devices **130** via a local system bus **150**. Various busses may be used to connect the central processing unit **101** to any of the I/O devices **130**, including a VESA VL bus, an ISA bus, an EISA bus, a MicroChannel Architecture (MCA) bus, a PCI bus, a PCI-X bus, a PCI-Express bus, and/or a NuBus, among others. For embodiments in which the I/O device is a video display **124**, the processor **101** may use an Advanced Graphics Port (AGP) to communicate with the display **124**. FIG. **1***d* depicts some embodiments of a computer **100** in which the main processor **101** communicates directly with I/O device

**130** via HyperTransport, Rapid I/O, or InfiniBand. FIG. **1***d* also depicts some embodiments in which local busses and direct communication are mixed: the processor **101** communicates with I/O device **130***a* using a local interconnect bus while communicating with I/O device **130***b* directly.

The computing device **121** may support any suitable installation device **116**, such as a floppy disk drive for receiving floppy disks such as 3.5-inch, 5.25-inch disks, or ZIP disks, a CD-ROM drive, a CD-R/RW drive, a DVD-ROM drive, tape drives of various formats, USB device, hard disk drive (HDD), solid-state drive (SSD), or any other device suitable for installing software and programs such as any client agent **120**, or portion thereof. The computing device **121** may further comprise a storage device **128**, such as one or more hard disk drives or solid-state drives or redundant arrays of independent disks, for storing an operating system and other related software, and for storing application software programs such as any program related to the client agent **120**. Optionally, any of the installation devices **116** could also be used as the storage device **128**. Additionally, the operating system and the software can be run from a bootable medium, for example, a bootable CD, such as KNOPPIX®, a bootable CD for GNU/Linux that is available as a GNU/Linux distribution from knoppix.net.

Furthermore, the computing device **121** may include a network interface **118** to interface to a Local Area Network (LAN), Wide Area Network (WAN) or the Internet through a variety of connections including, but not limited to, standard telephone lines, LAN or WAN links (e.g., T1, T3, 56 kb, X.25), broadband connections (e.g., ISDN, Frame Relay, ATM), wireless connections (e.g., IEEE 802.11), or some combination of any or all of the above. The network interface **118** may comprise a built-in network adapter, network interface card, PCMCIA network card, card bus network adapter, wireless network adapter, USB network adapter, modem, or any other device suitable for interfacing the computing device **121** to any type of network capable of communication and performing the operations described herein. A wide variety of I/O devices **130***a*-**130***n* may be present in the computing device **121**. Input devices include keyboards, mice, trackpads, trackballs, microphones, and drawing tablets, among others. Output devices include video displays, speakers, inkjet printers, laser printers, and dye-sublimation printers, among others. The I/O devices **130** may be controlled by an I/O controller **123** as shown in FIG. **1***c*. The I/O controller may control one or more I/O devices such as a keyboard **126** and a pointing device **127**, e.g., a mouse or optical pen. Furthermore, an I/O device may also provide storage **128** and/or an installation medium **116** for the computing device **121**. In still other embodiments, the computing device **121** may provide USB connections to receive handheld USB storage devices such USB flash drives.

In some embodiments, the computing device **121** may comprise or be connected to multiple display devices **124***a*-**124***n*, which each may be of the same or different type and/or form. As such, any of the I/O devices **130***a*-**130***n* and/or the I/O controller **123** may comprise any type and/or form of suitable hardware, software, or combination of hardware and software to support, enable, or provide for the connection and use of multiple display devices **124***a*-**124***n* by the computing device **121**. For example, the computing device **121** may include any type and/or form of video adapter, video card, driver, and/or library to interface, communicate, connect or otherwise use the display devices **124***a*-**124***n*. In some embodiments, a video adapter may comprise multiple connectors to interface to multiple display devices **124***a*-**124***n*. In some other embodiments, the computing device **121** may include multiple video adapters, with each video adapter connected to one or more of the display devices **124***a*-**124***n*. In some embodiments, any portion of the operating system of the computing device **121** may be configured for using multiple displays **124***a*-**124***n*. In some embodiments, one or more of the display devices **124***a*-**124***n* may be provided by one or more other computing devices connected to the computing device **121**, for example, via a network. Such embodiments may include any type of software designed and constructed to use another computer's display device as a second display device **124***a* for the computing device **121**. One ordinarily skilled in the art will recognize and appreciate the various ways and embodiments that a computing device **121** may be configured to have multiple display devices **124***a*-**124***n*.

In further embodiments, an I/O device **130** may be a bridge **170** between the system bus **150** and an external communication bus, such as a USB bus, an Apple Desktop Bus, an RS-232 serial connection, a SCSI bus, a FireWire bus, a FireWire **800** bus, an Ethernet bus, an AppleTalk bus, a Gigabit Ethernet bus, an Asynchronous Transfer Mode bus, a HIPPI bus, a Super HIPPI bus, a SerialPlus bus, a SCI/LAMP bus, a FibreChannel bus, and/or a Serial Attached small computer system interface bus, among others.

A computing device **121** of the sort depicted in FIGS. **1***c* and **1***d* may typically operate under the control of operating systems, which control scheduling of tasks and access to system resources. The computing device **121** can be running any operating system such as any of the versions of the Microsoft® Windows operating systems, any of the different releases of the Unix and Linux operating systems, any version of the Mac OS® for Macintosh computers, any embedded operating system, any real-time operating system, any open source operating system, any proprietary operating system, any operating systems for mobile computing devices, and/or any other operating system capable of running on a computing device and performing the operations described herein. Typical operating systems include: WINDOWS 3.x, WINDOWS 95, WINDOWS 98, WINDOWS 2000, WINDOWS NT 3.51, WINDOWS NT 4.0, WINDOWS CE, WINDOWS XP, WINDOWS VISTA, WINDOWS 7.0, WINDOWS SERVER 2003, and/or WINDOWS SERVER 2008, all of which are manufactured by Microsoft Corporation of Redmond, Wash.; MacOS, manufactured by Apple Computer of Cupertino, Calif.; OS/2, manufactured by International Business Machines of Armonk, N.Y.; and Linux, a freely-available operating system distributed by Red Hat of Raleigh, N.C., among others, or any type and/or form of a Unix operating system, among others.

In some embodiments, the computing device **121** may have different processors, operating systems, and input devices consistent with the device. For example, in one embodiment the computing device **121** is a Treo 180, 270, 1060, 600 or 650 smart phone manufactured by Palm, Inc. In this embodiment, the Treo smart phone is operated under the control of the PalmOS operating system and includes a stylus input device as well as a five-way navigator device. Moreover, the computing device **121** can be any workstation, desktop computer, laptop, or notebook computer, server, handheld computer, mobile telephone, any other computer, or other form of computing or telecommunications device that is capable of communication and that has sufficient processor power and memory capacity to perform the operations described herein.

Architecture

Reference is now made to FIG. **2**, which is a block diagram illustrating an architecture of a computing device **121** as discussed above regarding FIGS. **1***c* and **1***d*. The architecture of the computing device **121** is provided by way of illustration only and is not intended to be limiting. The architecture of computing device **121** may include a hardware layer **206** and a software layer divided into a user space **202** and a kernel space **204**.

Hardware layer **206** may provide the hardware elements upon which programs and services within kernel space **204** and user space **202** are executed. Hardware layer **206** also provides the structures and elements that allow programs and services within kernel space **204** and user space **202** to communicate data both internally and externally with respect to computing device **121**. The hardware layer **206** may include a processing unit **262** for executing software programs and services, a memory **264** for storing software and data, and network ports **266** for transmitting and receiving data over a network. Additionally, the hardware layer **206** may include multiple processors for the processing unit **262**. For example, in some embodiments, the computing device **121** may include a first processor **262** and a second processor **262'**. In some embodiments, the processor **262** or **262'** includes a multi-core processor. The processor **262** may include any of the processors **101** described above in connection with FIGS. **1***c* and **1***d*.

Although the hardware layer **206** of computing device **121** is illustrated with certain elements in FIG. **2**, the hardware portions or components of computing device **121** may include any type and form of elements, hardware or software, of a computing device, such as the computing device **121** illustrated and discussed herein in conjunction with FIGS. **1***c* and **1***d*. In some embodiments, the computing device **121** may comprise a server, gateway, router, switch, bridge, or other type of computing or network device, and have any hardware and/or software elements associated therewith.

The operating system of computing device **121** allocates, manages, or otherwise segregates the available system memory into kernel space **204** and user space **202**. As discussed above, in the example software architecture, the operating system may be any type and/or form of various ones of different operating systems capable of running on the computing device **121** and performing the operations described herein.

The kernel space **204** may be reserved for running the kernel **230**, including any device drivers, kernel extensions, and/or other kernel related software. As known to those skilled in the art, the kernel **230** is the core of the operating system, and provides access, control, and management of resources and hardware-related elements of the applications. In accordance with some embodiments of the computing device **121**, the kernel space **204** also includes a number of network services or processes working in conjunction with a cache manager sometimes also referred to as the integrated cache. Additionally, some embodiments of the kernel **230** will depend on embodiments of the operating system installed, configured, or otherwise used by the device **121**.

In some embodiments, the device **121** includes one network stack **267**, such as a TCP/IP based stack, for communicating with a client and/or a server. In other embodiments, the device **121** may include multiple network stacks. In some embodiments, the network stack **267** includes a buffer **243** for queuing one or more network packets for transmission by the computing device **121**.

As shown in FIG. **2**, the kernel space **204** includes a high-speed layer 2-7 integrated packet engine **240** and a policy engine **236**. Running packet engine **240** and/or policy engine **236** in kernel space **204** or kernel mode instead of the user space **202** improves the performance of each of these components, alone and in combination. Kernel operation means that packet engine **240** and/or policy engine **236** run in the core address space of the operating system of the device **121**. For example, data obtained in kernel mode may not need to be passed or copied to a process or thread running in user mode, such as from a kernel level data structure to a user level data structure. In this regard, such data may be difficult to determine for purposes of network application performance monitoring. In another aspect, the number of context switches between kernel mode and user mode are also reduced. Additionally, synchronization of and communications between packet engine **240** and/or policy engine **236** can be performed more efficiently in the kernel space **204**.

In some embodiments, any portion of the packet engine **240** and/or policy engine **236** may run or operate in the kernel space **204**, while other portions of packet engine **240** and/or policy engine **236** may run or operate in user space **202**. In some embodiments, the computing device **121** uses a kernel-level data structure providing access to any portion of one or more network packets, for example, a network packet comprising a request from a client or a response from a server. In some embodiments, the kernel-level data structure may be obtained by the packet engine **240** via a transport layer driver interface (TDI) or filter to the network stack **267**. The kernel-level data structure may include any interface and/or data accessible via the kernel space **204** related to the network stack **267**, network traffic, or packets received or transmitted by the network stack **267**. In some embodiments, the kernel-level data structure may be used by packet engine **240** and/or policy engine **236** to perform the desired operation of the component or process. Some embodiments provide that packet engine **240** and/or policy engine **236** is running in kernel mode **204** when using the kernel-level data structure, while in some other embodiments, the packet engine **240** and/or policy engine **236** is running in user mode when using the kernel-level data structure. In some embodiments, the kernel-level data structure may be copied or passed to a second kernel-level data structure, or any desired user-level data structure.

A policy engine **236** may include, for example, an intelligent statistical engine or other programmable application(s). In some embodiments, the policy engine **236** provides a configuration mechanism to allow a user to identify, specify, define or configure a caching policy. Policy engine **236**, in some embodiments, also has access to memory to support data structures such as lookup tables or hash tables to enable user-selected caching policy decisions. In some embodiments, the policy engine **236** may include any logic, rules, functions or operations to determine and provide access, control and management of objects, data or content being cached by the computing device **121** in addition to access, control and management of security, network traffic, network access, compression, and/or any other function or operation performed by the computing device **121**.

High speed layer 2-7 integrated packet engine **240**, also generally referred to as a packet processing engine or packet engine, is responsible for managing the kernel-level processing of packets received and transmitted by computing device **121** via network ports **266**. The high speed layer 2-7 integrated packet engine **240** may include a buffer for queuing one or more network packets during processing, such as for receipt of a network packet or transmission of a network packer. Additionally, the high speed layer 2-7 integrated packet engine **240** is in communication with one or more network stacks **267** to send and receive network packets via network ports **266**. The high speed layer 2-7 integrated packet engine **240** may work in conjunction with policy engine **236**. In particular, policy engine **236** is configured to perform functions related to traffic management such as request-level content switching and request-level cache redirection.

The high speed layer 2-7 integrated packet engine **240** includes a packet processing timer **242**. In some embodiments, the packet processing timer **242** provides one or more time intervals to trigger the processing of incoming (i.e., received) or outgoing (i.e., transmitted) network packets. In some embodiments, the high speed layer 2-7 integrated packet engine **240** processes network packets responsive to the timer **242**. The packet processing timer **242** provides any type and form of signal to the packet engine **240** to notify, trigger, or communicate a time related event, interval or occurrence. In many embodiments, the packet processing timer **242** operates in the order of milliseconds, such as for example 100 ms, 50 ms, or ms. For example, in some embodiments, the packet processing timer **242** provides time intervals or otherwise causes a network packet to be processed by the high speed layer 2-7 integrated packet engine **240** at a 10 ms time interval, while in other embodiments, at a 5 ms time interval, and still yet in further embodiments, as short as a 3, 2, or 1 ms time interval. The high speed layer 2-7 integrated packet engine **240** may be interfaced, integrated and/or in communication with the policy engine **236** during operation. As such, any of the logic, functions, or operations of the policy engine **236** may be performed responsive to the packet processing timer **242** and/or the packet engine **240**. Therefore, any of the logic, functions, and/or operations of the policy engine **236** may be performed at the granularity of time intervals provided via the packet processing timer **242**, for example, at a time interval of less than or equal to 10 ms.

In contrast to kernel space **204**, user space **202** is the memory area or portion of the operating system used by user mode applications or programs otherwise running in user mode. Generally, a user mode application may not access kernel space **204** directly, and instead must use service calls in order to access kernel services. As shown in FIG. **2**, user space **202** of computing device **121** includes a graphical user interface (GUI) **210**, a command line interface (CLI) **212**, shell services **214**, and daemon services **218**. Using GUI **210** and/or CLI **212**, a system administrator or other user may interact with and control the operation of computing device **121**. The GUI **210** may be any type and form of graphical user interface and may be presented via text, graphical or otherwise, by any type of program or application, such as a browser. The CLI **212** may be any type and form of command line or text-based interface, such as a command line provided by the operating system. For example, the CLI **212** may comprise a shell, which is a tool to enable users to interact with the operating system. In some embodiments, the CLI **212** may be provided via a bash, csh, tcsh, and/or ksh type shell. The shell services **214** may include the programs, services, tasks, processes and/or executable instructions to support interaction with the computing device **121** or operating system by a user via the GUI **210** and/or CLI **212**.

Daemon services **218** are programs that run continuously or in the background and handle periodic service requests received by computing device **121**. In some embodiments, a daemon service may forward the requests to other programs or processes, such as another daemon service **218** as appropriate. As known to those skilled in the art, a daemon service **218** may run unattended to perform continuous and/or periodic system wide functions, such as network control, or to perform any desired task. In some embodiments, one or more daemon services **218** run in the user space **202**, while in other embodiments, one or more daemon services **218** run in the kernel space.

## Collector Application

Reference is now made to FIG. **3**, which is a block diagram illustrating operations and/or functions of a collector application **200** as described above regarding FIG. **1***a*. The collector application **200** includes a kernel space module **310** and a user space module **320**. The kernel space module **310** may generally operate to intercept network activities as they occur. Some embodiments provide that the kernel space module **310** may use a kernel mode interface in the operating system, such as, for example, Microsoft Windows transport data interface (TDI). The kernel space module **310** may include a TDI filter **314** that is configured to monitor and/or intercept interactions between applications. Additionally, some embodiments provide that the kernel space module **310** may include an ancillary functions driver (AFD) filter **312** that is configured to intercept read operations and the time of their duration. Some operating systems may include a kernel mode driver other than the AFD. In this regard, operations described herein may be used with other such kernel mode drivers to intercept application operational data.

The raw data related to the occurrence of and attributes of transactions between network applications may be generally referred to as "performance data." The raw data may have value for diagnosing network application performance issues and/or for identifying and understanding the structure of the network applications. The measurements or aggregations of performance data may be generally referred to as "metrics" or "performance metrics." Performance data and the metrics generated therefrom may be temporally relevant (in other words, the performance data and the metrics may be directly related to and/or indicative of the health of the network at the time the performance data is collected). Performance data may be collected, and metrics based thereon may be generated, on a client side and/or a server side of an interaction. Some embodiments provide that performance data is collected in substantially real-time. In this context, "substantially real-time" means that performance data is collected immediately subsequent to the occurrence of the related network activity, subject to the delays inherent in the operation of the computing device and/or the network and in the method of collection. The performance data collected and/or the metrics generated may correspond to a predefined time interval. For example, a time interval may be defined according to the dynamics of the network and may include example period lengths of less than 1, 1, 5, 10, 15, 20, 30, and/or 60, seconds, among others.

Example client side metrics may be aggregated according to one or more applications or processes. For example, the client side metrics may be aggregated according to destination address, port number, and a local process identifier (PID). A PID may be a number used by some operating system kernels to uniquely identify a process. This number may be used as a parameter in various function calls allowing processes to be manipulated, such as adjusting the process's priority and/or terminating the process. In this manner, multiple connections from the same application or process to the same remote service may be aggregated.

Similarly, server side metrics may be aggregated according to the same application or service regardless of the client. For example, some embodiments provide that server side metrics may be aggregated according to local address, port number, and PID. Respective ones of the client side and server side metrics may be collected from the kernel space and/or user space.

The kernel space module **310** may include a kernel events sender **316** that is configured to receive performance data from the AFD filter **312** and/or the TDI filter **314**, and generate metrics based on the performance data for receipt by a kernel events receiver **322** in the user space module **320**. In the user space module **320**, metrics data received by the kernel event receiver **322** may be processed by a reverse domain name system (DNS) resolver **325** to map an observed network address to a more user-friendly DNS name. Additionally, metrics data received by the kernel events receiver **322** may be used by a process resolver **326** to determine the processes and/or applications corresponding to the collected kernel metrics data.

The user space module **320** may include a machine information collector **324** that is operable to determine static machine information, such as, for example, CPU speed, memory capacity, and/or operating system version, among others. As the performance data is collected corresponding to applications and/or processes, the machine information may be non-correlative relative to the applications and/or processes. The user space module **320** may include a process data collector **328** that collects data corresponding to the processes and/or applications determined in the process resolver **326**. A machine performance data collector **330** may collect machine specific performance data. Examples of machine data may include information about resource utilization such as the amount of memory in use and/or the percentage of available CPU time consumed. The user space module **320** may include an event dispatcher **332** that is configured to receive the machine information, resolved DNS information, process identification, process data, and/or machine data, and to generate events incorporating the aggregated metrics data for dispatch to a health data processor application **100** that is operable to receive aggregated metrics data from multiple collectors **200**.

Some embodiments provide that the performance data collected and/or metrics generated may be diagnostically equivalent and, thus, may be aggregated into a single event. The identification process may depend on which application initiates a network connection and which end of the connection is represented by a current collector application host.

Kernel level metrics may generally include data corresponding to read operations that are in progress. For example, reference is now made to FIG. **4**, which is a diagram illustrating determining a read wait time corresponding to a user transaction according to some embodiments of the present invention. A user transaction between a client **401** and a server **402** are initiated when the client **401** sends a write request at time T**1** to the server **402**. The server **402** completes reading the request at time T**2** and responds to the request at time T**3** and the client **401** receives the response from the server **402** at time T**4**. A kernel metric that may be determined is the amount of time spent between beginning a read operation and completing the read operation. In this regard, client measured server response time **410** is the elapsed time between when the request is sent (T**1**) and when a response to the request is read (T**4**) by the client. Accordingly, the client measured server response time **410** may be determined as T**4**−T**1**. The server **402** may determine a server measured server response time **412** that is the elapsed time between when the request is read (T**2**) by the server **402** and when the response to the request is sent (T**3**) by the server **402** to the client **401**. Accordingly, the server measured server response time **412** may be determined as T**3**−T**2**.

As the application response is measured in terms of inbound and outbound packets, the application response time may be determined in an application agnostic manner.

Additionally, another metric that may be determined is the read wait time **414**, which is the elapsed time between when the client **401** is ready to read a response to the request T**5** and when the response to the request is actually read T**4**. In some embodiments, the read wait time may represent a portion of the client measured server response time **410** that may be improved upon by improving performance of the server **402**. Further, the difference between the client measured server response time **410** and the server measured server response time **412** may be used to determine the total transmission time of the data between the client **401** and the server **402**. Some embodiments provide that the values may not be determined until a read completes. In this regard, pending reads may not be included in this metric. Further, as a practical matter, higher and/or increasing read time metrics discussed above may be indicative of a slow and/or poor performing server **402** and/or protocol where at least some messages originate unsolicited at the server **402**.

Other read metrics that may be determined include the number of pending reads. For example, the number of read operations that have begun but are not yet completed may be used to detect high concurrency. In this regard, high and/or increasing numbers of pending read operations may indicate that a server **402** is not keeping up with the workload. Some embodiments provide that the total number of reads may include reads that began at a time before the most recent aggregated time period.

Additionally, some embodiments provide that the number of reads that were completed during the last time period may be determined. An average of read wait time per read may be generated by dividing the total read wait time, corresponding to a sum of all of the T**4**-T**5** values during the time period, by the number of completed reads in that period.

In some embodiments, the number of stalled reads may be determined as the number of pending reads that began earlier than a predefined threshold. For example, a predefined threshold of 60 seconds may provide that the number of pending read operations that began more than 60 seconds ago are identified as stalled read operations. Typically, any value greater than zero may be undesirable and/or may be indicative of a server-initiated protocol. Some embodiments may also determine the number of bytes sent/received on a connection.

The number of completed responses may be estimated as the number of times a client-to-server message (commonly interpreted as a request) was followed by a server-to-client message (commonly interpreted as a response). Some embodiments provide that this may be measured by both the server and the client connections. In some embodiments, this may be the same as the number of completed reads for a given connection. Additionally, a total response time may be estimated as the total time spent in request-to-response pairs.

Reference is now made to FIG. **5**, which is a block diagram illustrating a kernel level architecture of a collector application **200** to explain kernel level metrics according to some embodiments of the present invention. As discussed above, regarding FIG. **3**, the collector may use a TDI filter **314** and an AFD filter **312**. The AFD filter **312** may intercept network activity from user space processes that use a library defined in a standard interface between a client application and an underlying protocol stack in the kernel.

The TDI filter **314** may operate on a lower layer of the kernel and can intercept all network activity. As the amount of information available at AFD filter **312** and TDI filter **314** is different, the performance data that may be collected and the metrics that may be generated using each may also be different. For example, the AFD filter **312** may collect AFD performance data and generate AFD metrics that include total read wait time, number of completed reads, number of pending reads and number of stalled reads, among others. The TDI filter may collect TDI performance data and generate TDI metrics including total bytes sent, total bytes received, total response time and the number of responses from the server. Depending on the architecture of a target application, the

AFD metrics for client-side connections may or may not be available. In this regard, if the application uses the standard interface, the collector may report non-zero AFD metrics. Otherwise, all AFD metrics may not be reported or may be reported as zero.

Some embodiments provide that kernel level metrics may be generated corresponding to specific events. Events may include read wait metrics that may include client side metrics such as total read wait time, number of completed reads, number of pending reads, number of stalled reads, bytes sent, bytes received, total response time, and/or number of responses, among others. Events may further include server response metrics such as bytes sent, bytes received, total response time and/or number of responses, among others.

In addition to the kernel metrics discussed above, the collector **200** may also generate user level metrics. Such user level metrics may include, but are not limited to aggregate CPU percentage (representing the percentage of CPU time across all cores), aggregate memory percentage (i.e., the percentage of physical memory in use by a process and/or all processes), and/or total network bytes sent/received on all network interfaces, among others. User level metrics may include, but are not limited to, the number of page faults (the number of times any process tries to read from or write to a page that was not in its resident in memory), the number of pages input (i.e., the number of times any process tried to read a page that had to be read from disk), and/or the number of pages output (representing the number of pages that were evicted by the operating system memory manager because it was low on physical memory), among others. User level metrics may include, but are not limited to, a queue length (the number of outstanding read or write requests at the time the metric was requested), the number of bytes read from and/or written to a logical disk in the last time period, the number of completed read/write requests on a logical disk in the last time period, and/or total read/write wait times (corresponding to the number of milliseconds spent waiting for read/write requests on a logical disk in the last time interval), among others.

Further, some additional metrics may be generated using data from external application programming interfaces. Such metrics may include, for example: the amount of memory currently in use by a machine memory control driver; CPU usage expressed as a percentage; memory currently used as a percentage of total memory; and/or total network bytes sent/received, among others.

In some embodiments, events may be generated responsive to certain occurrences in the network. For example events may be generated: when a connection, such as a TCP connection, is established from or to a machine; when a connection was established in the past and the collector application **200** first connects to the health data processing application **100**; and/or when a connection originating from the current machine was attempted but failed due to timeout, refusal, or because the network was unreachable. Events may be generated when a connection is terminated; when a local server process is listening on a port; when a local server process began listening on a port in the past and the collector application **200** first connects to the health data processing application **100**; and/or when a local server process ceases to listen on a port. Events may be generated if local network interfaces have changed and/or if a known type of event occurs but some fields are unknown. Events may include a description of the static properties of a machine when a collector application **200** first connects to a health data processing application **100**; process information data when a process generates its first network-related event; and/or information about physical disks and logical disks when a collector application **200** first connects to a health data processing application **100**.

Some embodiments provide that the different link events may include different data types corresponding to the type of information related thereto. For example, data strings may be used for a type description of an event. Other types of data may include integer, bytes and/or Boolean, among others.

In some embodiments, the events generated by collector application **200** for dispatch to heath data processing application **100** may incorporate metrics related to network structure, network health, computational resource health, virtual machine structure, virtual machine health, and/or process identification, among others. Metrics related to network structure may include data identifying the network device on which collector application **200** is executing, or data related to the existence, establishment, or termination of network links, or the existence of bound ports or the binding or unbinding of ports. Metrics pertinent to network health may include data related to pending, completed, and stalled reads, bytes transferred, and response times, from the perspective of the client and/or the server side. Metrics related to computational resource health may include data regarding the performance of the network device on which collector application **200** is executing, such as processing and memory usage. Metrics related to virtual machine structure may include data identifying the physical host machine on which collector application **200** is executing, and/or data identifying the virtual machines executing on the physical host machine. Metrics pertinent to virtual machine health may include regarding the performance of the host machine and/or the virtual machines executing on the host machine, such as processing and memory usage as determined from the perspective of the host machine and/or the virtual machines. Finally, metrics related to process identification may include data identifying individual processes executing on a network device.

Reference is made to FIG. **6**, which illustrates example operations that may be carried out by collector application **200** in monitoring and reporting network application performance according to some embodiments of the present invention. At block **600**, collector application **200** establishes hooks on a networked device to an internal network protocol kernel interface utilized by the operating system of the networked device. In some embodiments, these hooks may include, for instance, a TDI filter, Collector application **200** also establishes hooks to an application oriented system call interface to a transport network stack. The hooks may include, in some embodiments, an AFD filter. Collector application **200** collects, via the established hooks, performance data corresponding to at least one network application running on the networked device (block **602**). At block **604**, kernel level and user level metrics are generated based on the collected performance data. The generated metrics may provide an indication of the occurrence of an interaction (e.g., establishment of a network link), or may provide measurements of, for instance, a count of some attribute of the collected performance data (e.g., number of completed reads) or a summation of some attribute of the collected performance data (e.g., total read attempts). The kernel level and user level metrics are aggregated by application—e.g., by aggregating metrics associated with the same IP address, local port, and process ID (block **606**). At block **608**, the kernel level and user level metrics generated within a specified time interval are aggregated. For instance, in some embodiments, metrics generated within the most recent 15-second time interval are aggregated.

At block **610**, redundant data is removed from the aggregated metrics, and inconsistent data therein is reconciled.

Redundant data may include, for instance, functionally equivalent data received from both the TDI and AFD filters. Collector application **200** performs a reverse DNS lookup to determine the DNS name associated with IP addresses referenced in the generated kernel level and user level metrics (block **612**). Finally, at block **614**, an event is generated, incorporating the kernel level and user level metrics and the determined DNS name(s). The generated event may be subsequently transmitted to health data processing application **100** for incorporation into a model of network health status.

Installation without Interruption

In some embodiments, the collector application **200** may be installed into a machine of interest without requiring a reboot of the machine. This may be particularly useful in the context of a continuously operable system, process and/or operation as may be frequently found in manufacturing environments, among others. As the collector operations interface with the kernel, and more specifically, the protocol stack, installation without rebooting may include intercepting requests coming in and out of the kernel using the TDI filter. Some embodiments include determining dynamically critical offsets in potentially undocumented data structures. Such offsets may be used in intercepting network activity for ports and connections that exist prior to an installation of the collector application **200**. For example, such previously existing ports and connections may be referred to as the extant state of the machine.

Some embodiments provide that intercepting the stack data may include overwriting the existing stack function tables with pointers and/or memory addresses that redirect the request through the collector filter and then to the intended function. In some embodiments, the existing stack function tables may be overwritten atomically in that the overwriting may occur at the smallest indivisible data level. Each entry in a function table may generally include a function pointer and a corresponding argument. However, only one of these entries (either the function or the argument) can be overwritten at one time. Thus, intercepting function calls may rely on two consecutive overwrites of the stack data corresponding to the function and corresponding argument. In some embodiments, there is no means for protecting from an intervening operation between overwriting one of the function and argument and overwriting the other one of them. In this regard, system stability may be at risk from two attempted consecutive overwrites.

As the consecutive overwrites of intercepting function calls may place the machine at risk of instability, a dynamic overwriting operation may be used. Specifically, a separate data structure is provided that includes a pointer to the original function, its original argument and dynamically generated code to call a filter in the collector application **200**. The address of this data structure may be used to atomically overwrite the original function pointer in a single operation. The collector collects the data and then calls the original function corresponding to the overwritten stack data to perform its intended purpose. In this manner, the original behavior of the machine is preserved and the collector application collects the relevant data without rebooting the machine and/or placing the machine at risk of instability.

Some embodiments may include identifying the potentially undocumented data structures representing bound ports and network connections. For example, TDI objects (connections and bound ports) created prior to the installation of the collector application **200** may be determined by first enumerating all objects identified in a system. Each of the enumerated objects may be tagged with an identifier corresponding to its sub-system. A request corresponding to a known TDI object is created and sent for processing. The type codes of the enumerated objects are compared to those of the known TDI object to determine which of the objects are ports and which of the objects are connections. The enumerated objects may then be filtered as either connections or ports.

In some embodiments, this may be accomplished using an in-kernel thread. The thread may monitor network connections having restricted visibility and may detect when a monitored connection no longer exists. Connections may be added dynamically to the monitored list as needed.

Some embodiments provide that events may be generated to indicate that visibility into network events may be incomplete. For example, information may be missing corresponding to an active process, the state of a known connection, and/or missing information regarding network activity. In this manner, depending on conditions, a custom event can be transmitted to indicate what type of information is missing and what process may be responsible for that information.

Health Data Processing Application

In some embodiments, the health data processing application **100** may be operable to receive, from at least one collector application **200**, network activity data corresponding to network activity of the applications on the network device on which the collector application **200** is installed. The health data processing application **100** may combine the network activity data received from the collector application **200** to remove redundant portions thereof. In some embodiments, the health data processing application **100** may archive the received activity data in a persistent data store along with a timestamp indicating when the activity data was collected and/or received. The health data processing application **100** may generate a model that includes identified network application components and their relatedness and/or links therebetween. The generated model may be displayed via one or more display devices such as, e.g., display devices **124*a*-124*n*** discussed in greater detail above.

In some embodiments, the health data processing application **100** may be operable to combine network activity data reported from multiple collector applications **200** to eliminate redundancy and to address inconsistencies among data reported by different collector applications **200**. For example, network data from multiple collector applications **200** may be stitched together to create a consistent view of the health of the network applications.

Some embodiments provide that the model may be a graphical display of the network including application components (machines, clients, processes, etc.) and the relationships therebetween. In some embodiments, the model may be generated as to reflect the real-time or near-real-time activity of the network. It is to be understood that, in this context, "near-real-time" may refer to activity occurring in the most recent of a specified time interval for which activity data was received. For instance, health data processing application **100** may receive from collector applications **200** aggregated activity data corresponding to the most recent 15-second interval of network operation, and, accordingly, the model of near-real-time activity may reflect the activity of the network as it existed during that most recent 15-second interval.

Some embodiments provide that the model may be generated to reflect an historical view of network activity data corresponding to a specified time interval. The historical view may be generated based on archived activity data retrieved from a persistent data store and having a timestamp indicating that the activity data was collected or received during the specified time interval. In other embodiments, the model may be dynamically updated to reflect new and/or lost network collectors and/or network components. Further, graphs may be provided at each and/or selected network resource indicators to show activity data over part of and/or all of the time interval.

In some embodiments, a model may include sparklines to provide quick access to trends of important metrics, process and application views to provide different levels of system detail, and/or model overlays to provide additional application analysis. For example, visual feedback regarding the contribution of a network link relative to a given criterion may be provided. In this manner, hop by hop transaction data about the health of applications can be provided. Additionally, visual ranking of connections based on that criteria may be provided. Bottleneck analysis based on estimated response times may be provided to identify slow machines, applications, and/or processes, among others.

Some embodiments provide that health data processing application **100** may be operable to infer the existence of network devices and/or network applications for which no activity data was received or on which no collector application **200** is running, based on the identification of other network devices and/or other network applications for which activity data was received. For instance, activity data received by health data processing application **100** may indicate that a network link has been established between a local network device running collector application **200** and a remote network device that is not running collector application **200**. Because the activity data may include identifying information for both the local and remote network devices, health data processing application **100** may infer that the remote network device exists, and incorporate the remote network device into the generated model of network activity.

In other embodiments, health data processing application **100** may be operable to identify a network application based on predefined telecommunications standards, such as, e.g., the port numbers list maintained by the Internet Assigned Numbers Authority (IANA). Health data processing application **100** may, for example, receive activity data indicating that a process on a network device is bound to port **21**. By cross-referencing the indicated port number with the IANA port numbers list, health data processing application **100** may identify the process as an File Transfer Protocol (FTP) server, and may include the identification in the generated model.

Reference is made to FIG. **7**, which is a screen shot of a graphical user interface (GUI) including a model generated by a health data processing application according to some embodiments of the present invention. The GUI **700** includes a model portion **701** that illustrates representations of various network applications and/or application components **702**. Such representations may include identifier fields **704** that are operable to identify application and/or application component addresses, ports, machines and/or networks. Connections **706** between network applications and/or application components may be operable to convey additional information via color, size and/or other graphical and/or text-based information. A summary field **708** may be provided to illustrate summary information corresponding to one or more applications and/or application components, among others. A port identification portion **712** may be operable to show the connections corresponding to and/or through a particular port. The GUI **700** may include a system and/or network navigation field **710**, overlay selection field **714**, and one or more time interval and/or snapshot field(s) **716**.

FIG. **8** is a flowchart illustrating example operations that may be carried out by health data processing application **100** in generating and displaying a real-time model of network application health according to some embodiments of the present invention. At block **800**, health data processing application **100** may receive activity data from a plurality of collector applications **200** executing on respective ones of a plurality of network devices. The received activity data corresponds to activities of a plurality of network applications executing on respective ones of the plurality of networked devices. At block **802**, the received activity data is archived along with a timestamp indicating when the activity data was collected and/or received. As discussed in greater detail with respect to FIG. **9**, this archived data may allow health data processing application **100** to generate and display an historical model of network application health during a specified time interval. At block **804**, the received activity data is combined to remove redundant data and to reconcile inconsistent data. At block **806**, health data processing application **100** identifies the network applications executing on the respective ones of the plurality of networked devices, and ascertains the relationships therebetween. The identification of the network applications and the relationships therebetween may be based on the received activity data, and may further be determined based on a correlation between the received activity data and predefined industry standards, as discussed above. At block **808**, health data processing application **100** may infer the existence of network applications for which no activity data was received, based on the identification of network applications for which activity data was received. At block **810**, a real-time model of network health status, including the identified network applications and the relationships therebetween, is generated, and the model is displayed at block **812**.

FIG. **9** is a flowchart illustrating example operations carried out by a health data processing application **100** in generating and displaying an historical model of network application health according to some embodiments of the present invention. At block **900**, the activity data previously archived at block **802** and corresponding to a specified time interval is retrieved. The retrieved activity data is combined to remove redundant data and reconcile inconsistent data at block **902**. At block **904**, health data processing application **100** identifies the network applications associated with the retrieved activity data, and ascertains the relationships therebetween. The identification of the network applications and the relationships therebetween may be based on the retrieved activity data, and may further be determined based on correlation between the retrieved activity data and industry standards. At block **906**, health data processing application **100** may infer the existence of network applications for which no activity data was retrieved, based on the identification of network applications for which activity data was retrieved. At block **908**, an historical model of network health status in the specified time interval, including the identified network applications and the relationships therebetween, is generated, and the historical model is displayed at block **910**.

Custom Protocol

Some embodiments provide that transferring the activity data between the collector applications **200** and the health data processing application **100** may be performed using a compact, self-describing, linear buffer communications protocol. In some embodiments, the custom protocol uses a common representation for monitoring information, commands and configuration data. As the methods and systems described herein are intended to monitor network performance, the protocol may be operable to minimize the volume of information exchanged between the collector applications **200** and the health data processing application **100**.

In some embodiments, the collector applications **200** are operable to generate events in a streaming data format. Events may be generated corresponding to the predefined monitoring time period. Information provided corresponding to an event may include an event type, network resource identification data including PID, remote identifiers, quantities and/or types of data sent/received, and/or response time information, among others. The protocol may include a banner portion that may be established through a handshaking process that may occur when a collector application **200** initially communicates with the health data processing application **100**. The banner portion may define the data types and formats to be transferred. In this manner, the protocol may be flexible by virtue of the self-descriptive banner portion and may avoid sending unused, unwanted or blank data fields.

Monitoring the Contents of Network Traffic in a Network Device

As discussed above, collector application **200** may collect raw data related to the occurrence and attributes of transactions between network applications (i.e., "performance data"), which may have value for diagnosing network application performance issues and/or for identifying and understanding the structure of the network applications. Collector application **200** may also generate events that provide measurements or aggregations of performance data (i.e., "metrics" or "performance metrics"). However, some performance issues, particularly those that occur sporadically, or those for which identifying and/or recreating the circumstances leading to the performance issues is difficult, may be challenging to diagnose using performance data and metrics. In such scenarios, identification and analysis of transaction data that is communicated between network applications and that is contained within the raw network traffic data may provide diagnostic value.

In some embodiments, therefore, collector application **200** may provide a method for parsing and optionally filtering network traffic data to identify and extract transaction data contained therein. FIG. **10** illustrates the architecture of an example system providing network traffic data parsing and filtering according to some embodiments of the present invention. As detailed above with respect to FIG. **3**, kernel space module **310** in kernel space **204** is operative to intercept packets of network traffic data, in substantially real-time, as the data is sent by a network device and/or received by a network device. In this context, "substantially real-time" means that network traffic data is collected immediately subsequent to the sending and/or receiving of the network traffic data, subject to the delays inherent in the operation of the computing device and/or the network and in the method of collection.

Kernel space module **310** is communicatively coupled to transactions virtual machine (TVM) **1000**, which may be consulted by kernel space module **310** with respect to each packet of network traffic data to determine whether the packet should be collected and sent for parsing. In some embodiments, TVM **1000** may provide a compact and efficient bytecode representation of a criteria predicate expression (for example, a Boolean combination of individual predicates) that defines the characteristics of network traffic data to be collected. For instance, a criteria predicate may dictate that only network traffic data arriving on a particular network port, network traffic data sent from a particular remote IP address, and/or network traffic data sent or received by a particular application is to be collected. In this way, TVM **1000** may effectively describe "interesting" network traffic data that is to be collected.

Network traffic data deemed "interesting" by TVM **1000** is collected by kernel space module **310** and sent to shared memory buffer **1005**. Shared memory buffer **1005** is an area of memory that accessible by both kernel space module **310** executing in kernel space **204** and other modules of collector application **200** executing in user space **202**. By using shared memory buffer **1005**, kernel space module **310** in some embodiments may incur few or no context switches in transferring the collected network traffic data, thus allowing large amounts of data to be transferred to collector application **200** efficiently and asynchronously. Some embodiments may provide that the size of shared memory buffer **1005** may be configurable by, for instance, a configuration file. In some embodiments, the size of shared memory buffer **1005** may be adaptive based on available memory. For example, if a surplus of memory becomes available, the size of shared memory buffer **1005** may be automatically increased, whereas if the amount of available memory is reduced, the size of shared memory buffer **1005** may be automatically decreased.

Parser engine **1010** of collector application **200** executing in user space **202** retrieves collected network traffic data from shared memory buffer **1005** and determines how the collected network traffic data will be further processed. In some embodiments, parser engine **1010** may pass the collected network traffic data into one or more protocol-specific parsers for processing, depending on the network protocol of the collected network traffic data. For example, protocol-specific parsers **1015**, **1020**, **1025**, and **1030** may be associated with the Oracle Structured Query Language (SQL), Microsoft SQL (MS-SQL), Message Queue (MQ), and LDAP network protocols, respectively. Thus, if the collected network traffic data is associated with the Oracle SQL query protocol, then the collected network traffic data may be passed to Parser A **1015** for processing. Likewise, collected network traffic data associated with a MS-SQL query may be passed to Parser B **1020**, collected network traffic data associated with a MQ query may be forwarded to Parser C **1025**, and collected network traffic data associated with an LDAP query may be sent to Parser D **1030**. The protocol-specific parsers **1015**, **1020**, **1025**, and **1030** may extract transaction data related to logical transactions defined by the respective network protocols. It is to be understood that the network protocol processed by an example protocol-specific parser may be any network protocol for which network traffic data may be collected, and is not limited to the network protocols enumerated above. It is to be further understood that parser engine **1010** may forward the same collected network traffic data to multiple protocol-specific parsers, or to no parser.

The transaction data related to logical transactions and extracted by a protocol-specific parser may include diagnostically useful associated metadata. In some embodiments, the associated metadata may include, for example, server response time, the size in bytes of the request and/or the response, an error code (if a transaction was unsuccessful), a timestamp associated with the request, the IP address of the client and/or the server, the server port, and/or the server process ID, among others. The associated metadata may also include protocol-specific metadata extensions, including user name, database name, and/or session ID, among others.

In some embodiments, protocol-specific parsers include shared libraries, each implementing a well-defined parser Application Programming Interface (API), and each loaded at run time by collector application **200** according to the contents of a configuration file. Each protocol-specific parser, in some embodiments, may have access to the configuration and logging facilities of collector application **200**.

Some embodiments may provide a protocol-specific parser that implements the parser API, and that further incorporates an embedded script interpreter that is operable to execute a script that defines parsing operations. For example, Parser D **1030** may include a module implementing the parser API, and may have embedded within it parser script interpreter **1035**, which interprets a script to determine how collected network traffic data passed to Parser D **1030** will be parsed. Parser script interpreter **1035** may be operable to interpret scripting languages such as Perl, PHP, JavaScript, Ruby, Python, and/or Tel, among others. The use of an interpreted script to define parsing operations may allow for the quick prototyping of protocol-specific parsers, as well as providing a simplified means for support personnel and users to implement protocol-specific parsers. Protocol-specific parsers using embedded script interpreters may also provide platform independence, and may permit parser developers to take advantage of the inherent capabilities of the particular scripting language used.

Each protocol-specific parser **1015**, **1020**, **1025**, and **1030** may "consume" more or less data than is made available to it by parser engine **1010**. In this context, "consume" means to indicate that a specified quantity of data has been processed. For example, a protocol-specific parser may examine the collected network traffic data, and may be able to determine based on the collected network traffic data that a subsequent portion of network traffic data is likely to be unimportant or irrelevant. The collected network traffic data, for instance, may contain a transaction identifier, from which the protocol-specific parser may determine that the subsequent portion of network traffic data contains only padding or other data of no interest. Accordingly, the protocol-specific parser may indicate that the collected network traffic data and the subsequent portion of network traffic data has been consumed (i.e., that the protocol-specific parser has consumed more data than was actually made available to it by parser engine **1010**). Conversely, the protocol-specific parser may determine that part of the available collected network traffic data is not sufficient to extract a complete transaction, or that the available collected network traffic data is insufficient to extract even a single logical transaction. In this case, the protocol-specific parser may indicate that only some, or none, of the collected network traffic data was consumed (i.e., that the protocol-specific parser has consumed less data than is available). In some embodiments, this may allow the collected network traffic data to be buffered until sufficient data has accumulated to permit the extraction of one or more complete logical transactions by the protocol-specific parser. A protocol-specific parser, in some embodiments, may determine that the network traffic data collected from a particular network flow cannot be parsed at all, and, in response, will indicate that subsequent network traffic data corresponding to the same network flow is not to be parsed.

After a protocol-specific parser has extracted transaction data corresponding to at least one logical transaction from the collected network traffic data, it may pass the extracted transaction data directly to parser sink **1060**, as, for example, with Parser D **1030** in FIG. **10**. In some embodiments, a protocol-specific parser may forward the extracted transaction data to a filter for further processing. For example, parsers **1015**, **1020**, and **1025** may pass extracted transaction data to filters **1040**, **1045**, and **1050**, respectively. It is to be understood that there may be any number of filter stages following a protocol-specific parser—i.e., the output generated by one filter may be passed to another filter in a linear fashion, thus creating a "pipeline" sequence of parsing and filtering modules. It is to be further understood that a filter may receive input from more than one protocol-specific parser.

Each filter may alter the transaction data passed into it by modifying and/or deleting parts of the data, or by supplementing the transaction data with additional data. For instance, filters **1040** and **1045**, which may receive SQL transaction data extracted by parsers **1015** and **1020**, respectively, may be

operable to normalize SQL transaction data by removing string and numeric literals and extraneous whitespace, and by capitalizing keywords within the SQL transaction data passed into the filters. In some embodiments, a filter may identify and aggregate related logical transactions, and may represent the aggregated logical transactions as a single filtered transaction. For example, a filter may create and output a logical transaction representing the combination of all Uniform Resource Locator (URL) transactions from a single client corresponding to retrieval of one complete web page.

As with protocol-specific parsers, filters, in some embodiments, may incorporate an embedded script interpreter that is operable to execute a script that defines filtering operations. For example, Filter C **1050** has embedded within it filter script interpreter **1055**, which interprets a script to determine how extracted transaction data passed to Filter C **1050** will be filtered. Filter script interpreter **1055** may be operable to interpret scripting languages such as Perl, PHP, JavaScript, Ruby, Python, and/or Tel, among others.

Filters **1040**, **1045**, and **1050** and protocol-specific parser **1030** pass their respective outputs to parser sink **1060** of collector application **200**. Parser sink **1060** may aggregate the transaction data that was extracted and/or filtered within a predefined time interval, and may then generate an event based on the aggregated transaction data. In some embodiments, health data processing application **100** may receive events from collector application **200**, and may request that collector application **200** data send transaction data in "trace mode." In trace mode, parser sink **1060** may aggregate the transaction data, as above, and also may compress the entirety of the transaction data and generate an event based on the compressed transaction data. This may provide health data processing application **100** with more detailed transaction data for use in diagnosing network and/or application performance issues.

Reference is now made to FIG. **11**, which illustrates example operations carried out by a kernel space module of collector application **200** in collecting raw network traffic data to be parsed and/or filtered. At block **1100**, kernel space module **310**, using a kernel space driver interface, collects network traffic data sent by and/or received at a network device. Kernel space module **310** determines whether the collected network traffic satisfies a criteria predicate, which may specify the characteristics of the network traffic data for which parsing and/or filtering is to be carried out (block **1105**). In some embodiments, for example, the criteria predicate may specify that only network traffic data that arrives on a particular port, network traffic data that was sent from a particular remote IP address, and/or network traffic data that was sent or received by a particular program is to be parsed and/or filtered. If the collected network traffic data does not satisfy the criteria predicate, the collected network traffic data is disregarded (block **1110**), and kernel space module **310** resumes operation again at block **1100**. If the collected network traffic data does satisfy the criteria predicate, then kernel space module **310** transfers the collected network traffic data into a shared memory buffer **1005** (block **1115**). As detailed above with respect to FIG. **10**, shared memory buffer **1005** is an area of memory that is shared between and accessible by both kernel space module **310** executing in kernel space **1000** and other modules of collector application **200** executing in user space **1005**. After the collected network traffic data is transferred to shared memory buffer **1005**, the kernel space module **310** resumes operation at block **1100**.

FIG. **12** illustrates example operations carried out by collector application **200** in parsing and/or filtering the collected network traffic data. At block **1200**, collector application **200** determines whether any collected network traffic data is available to be processed in shared memory buffer **1005**. If there is no collected network traffic data to be processed, then operation resumes at block **1200**. If shared memory buffer **1005** contains collected network traffic data, collector application **200** uses an appropriate protocol-specific parser to examine the collected network traffic data and determine whether the data can be parsed (block **1205**). If the data cannot be parsed—such as, for example, the collected network traffic data is in a format not recognized by the protocol-specific parser, or is otherwise incomprehensible—then collector application **200** stores an indicator signifying that no subsequent network traffic data from the corresponding network source will be parsed (block **1210**), and operation resumes at block **1200**.

If the data can be parsed, the protocol-specific parser examines the collected network traffic data to determine whether parsing of subsequent network traffic data is necessary (blocks **1215** and **1220**). For example, the collected network traffic data may contain a transaction identity indicating that a subsequent portion of network traffic data contains only padding or other data of no interest. If further parsing is unnecessary, the protocol-specific parser stores an indicator indicating that both the collected network traffic data and the subsequent portion of network traffic data were consumed, and operation resumes at block **1200**.

If further parsing is necessary, then the protocol-specific parser examines whether the collected network traffic data is sufficient to allow the extraction of at least one logical transaction (block **1230**). If there is not enough collected network traffic data to extract at least one logical transaction (as may be the case, for instance, where the collected network traffic data comprises a single IP packet, but a logical transaction spans multiple IP packets), then the protocol-specific parser stores an indicator indicating that none of the collected network traffic data was consumed (block **1235**). In some embodiments, this permits the collected network traffic data to be buffered until sufficient data has been collected to enable the extraction of at least one logical transaction. Operation then resumes at block **1200**.

If sufficient network traffic data has been collected, then the protocol-specific parser extracts transaction data corresponding to at least one logical transaction that is defined by a network protocol from the collected network traffic data (block **1240**). For example, depending on the network protocol associated with the collected network traffic data, transaction data may include URL of a requested web page or the contents of a SQL, LDAP, and/or MQ query, among others. At block **1245**, the protocol-specific parser stores an indicator indicating the actual quantity of data that was consumed. An attribute of the extracted transaction data is stored in memory and/or in a persistent data store (block **1250**). In some embodiments, this may allow the protocol-specific parser to maintain state information regarding the logical transactions for which data is extracted from the collected network traffic data.

At block **1255**, a filter generates filtered transaction data based on the extracted transaction data passed to it from the protocol-specific parser. As detailed above with respect to FIG. **10**, a filter may modify and/or delete data in the extracted transaction data, and/or may supplement the extracted transaction data with additional data. An attribute of the filtered transaction data is stored in memory and/or in a persistent data store (block **1260**). In some embodiments, this may allow the filter to maintain state information regarding the logical transactions represented by the filtered transaction data.

Collector application **200** aggregates the filtered transaction data that was extracted during a predefined time interval (block **1265**) and generates an event based on the aggregated data (block **1270**). Collector application **200** then determines whether a trace mode has been requested by health data processing application **100** (block **1275**). If so, then the entire contents of the filtered transaction data that was extracted during a predefined time interval is compressed (block **1280**), and an additional event is generated by collector application **200** based on the compressed data (block **1285**). Operation then resumes at block **1200**.

Correlation Analysis of Collected Metrics

As described above, the Collector application **200** generates performance data or metrics related to and/or indicative of the health of the network. Large numbers of metrics about a distributed application may be collected, including user-visible performance metrics such as transactions per second and latency of transactions as well as infrastructure metrics relating to CPU, memory, and disk load. Users that are presented with these numerous metrics may desire to identify applications and/or transactions that are being processed slowly in this network, which machine-to-machine contexts are slowing traffic in the system, or which infrastructure resources are limiting performance. In other words, users would like to discover relationships among the various metrics that have been collected that indicate reasons for performance degradation.

Reference is made to FIG. **13**, which illustrates example operations carried out by a correlation analyzer that is operable to analyze correlations of collected network traffic data metrics, according to some embodiments of the present invention. A set of collected metrics are available that include network element metrics and/or node-to-node contexts, among others. At block **1301**, the correlation analyzer selects a primary metric from the set of collected metrics. The primary metric, which may also be referred to as the correlated metric, is the metric to which other data is correlated. This primary metric may be selected by the user using a user interface and/or may be automatically selected by the application based on configured criterion or through analysis regarding the health of the network by identifying points of interest within the network.

As shown in block **1303**, correlation coefficients between the primary metric and ones of at least a portion of the plurality of the collected metrics are generated. The correlation coefficient, according to some embodiments, may be calculated using Pearson's correlation coefficient. Other types of correlation coefficients well known in the art, such as Spearman's rank correlation may also be used. Correlation coefficients typically are in the range −1.0 to 1.0. For example, a correlation coefficient of 1.0 signifies that a data pair is identical, or that one is a positive number multiple of the other. Usually, a correlation coefficient near 1.0 signifies strong correlation between the data pair. A correlation coefficient of −1.0 signifies inverse correlation between the data pair, or that one is a negative number multiple of the other. Correlation coefficients between 0 and 1.0 signify varying degrees of correlation. Similarly, correlation coefficients between −1.0 and 0 signify varying degrees of inverse correlation. Correlation coefficients near 1.0 indicate strong correlation while correlation coefficients near −1.0 indicate strong inverse correlation.

A hypothesis set may be generated based on the correlation coefficients, as shown in block **1304**. A hypothesis set may include metrics likely to be causally related to the primary metric. A hypothesis may include metrics whose behavior influences/causes or whose behavior is influenced/caused by the primary metric. Determining the hypothesis set may include identifying the primary metric and one or more of respective collected metrics for which the respective correlation coefficients are greater than a first threshold or less than a second threshold (block **1305**). The correlation coefficient being greater than the first threshold may correspond to a positive correlation coefficient. Similarly, the correlation coefficient being less than the second threshold may correspond to a negative correlation coefficient. These identified metrics, qualified by the aforementioned thresholds may be used to determine the primary metric and the respective collected metrics that may be added to the hypothesis set, as shown in block **1306**. For example, a first threshold value of 0.7 and a second threshold value of −0.7 may be selected. The hypothesis set for these example thresholds would include metric pairs which have a positive correlation greater than 0.7 or an inverse correlation less than −0.7. A positive correlation, for example, may suggest that higher load on one network element or context may be causally linked to increased utilization of another network element or resource. Similarly, a negative correlation may suggest, for example, that one process or transaction may be causally linked to inhibiting another process or transaction from obtaining adequate resources.

Additionally, as illustrated by block **1307**, heat maps based on the correlation coefficients may be generated. Heat maps may use color-coded variations to distinguish varying levels of correlation between the between the primary metric and the collected metrics. The heat maps may be useful to the user to identify hot spots in the network that show high correlation among certain metric pairs. A heat map may be generated for correlation coefficients including all pairs of the collected metrics and/or for one or more subsets of all of the pairs of collected metrics.

As shown in block **1302**, defined settings may be used to determine a candidate set which corresponds to a portion of the plurality of collected metrics to which the primary metric may be correlated. Reference is now made to FIG. **14** which is a flowchart illustrating example settings that may be operations that may be included in identifying a candidate set (block **1302**), according to some embodiments of the present invention. As illustrated in block **1401**, a scope of network elements to include in the candidate set may be defined. The scope may be defined by a number of network element hops from the selected primary metric's network element. For example, the candidate set may include all collected metrics associated with network elements that are N hops from the network element associated with the primary metric, where N>=1. In some embodiments, the scope may be defined to include all network elements that are associated with the network element of the primary metric. For example, the associated network elements may include all other network elements that have common data paths, share memory, and/or share processes with the network element of the primary metric, among others.

The candidate set may also be defined by selecting a network metric type that identifies the types of network metrics to include in the candidate set, as illustrated in block **1402** of FIG. **14**. Example network metric types that may be used to determine the candidate set include transactions, virtual machines, infrastructure metrics, links between nodes in the network, processes running on nodes in the network, server stacks, threads, applications, and memory usage such as cache, stack, virtual memory, and storage, among others.

Once the network metric types are selected, as shown in block **1403**, a filtering function may be applied to the collected metrics. As further illustrated in block **1404**, applying the filter function may include receiving an input associated with the network metric type. Reference is now made to FIG. **16**, which is a flowchart illustrating example operations that generate the network metric type for use in identifying the candidate set. The received input may be from a user via a user input as in **1601**, where the network metric type is generated based on the input from the user, as shown in **1602**. In some embodiments, the input associated with the network metric type may be obtained by reading a data file that includes configuration information (block **1603**). The configuration information may be used to generate the network metric type (block **1604**). The received input may also be from other applications, modules, threads, or processes that may determine the information that may be relevant to the operator utilizing the network health monitoring functionality. Referring once again to FIG. **14**, block **1405** illustrates that the members of the candidate set are identified out of the collected metrics based on the received input.

In some embodiments, transformations may be applied to the data before calculating correlation values. Example transformations may include smoothing the data, curve fitting, and/or time-shifting, among others. Smoothing a data set may include creating an approximating function that attempts to capture important patterns in the data, while leaving out noise or other fine-scale structures. Outlier data points may be removed or modified in order to provide a smoother data set. In smoothing, the data points may be modified so that individual high points (presumably because of noise) are reduced, and points that are lower than the adjacent points are increased leading to a smoother data set. Many different algorithms may be used for smoothing, one example being the histogram.

Reference is now made to FIG. **15** which illustrates another example of a transformation that may be applied to the data by applying temporal shifting of relative metrics during correlation analysis. In general, primary metric and collected metric data pairs are selected for analysis from the same time interval. When applying temporal shifting, the primary metric and the respective collected metric to be correlated are time shifted with respect to one another such that they are from different time intervals. As used herein, temporal shifting and time shifting refer to any selection of members of a data pair from different time intervals with respect to one another. In some embodiments, the user may control the temporal shift as described in blocks **1501**, **1502**, and **1503**. The user interface may provide a mechanism by which the user may select to increment and/or decrement the input (block **1501**), and correspondingly adjust the time interval based on this received increment/decrement (block **1502**). In some embodiments shown in block **1503**, the correlation analyzer may receive a temporal shift value from the user interface and adjust the time interval based on this received temporal shift value. The time interval used for temporal shifting may be normalized to the time granularity of the network monitoring application. In other words, the time interval may be a multiple of the data collection interval corresponding to the collected metrics. Based on the user specified time interval, the data corresponding to the primary metric corresponds to a first time while the data corresponding to the portion of the collected metrics corresponds to a second time that is different from the first time by the user specified time interval. As further illustrated in FIG. **15** blocks **1301**, **1303**, and **1304**, correlation analysis is performed on this time-shifted data such that a temporally shifted hypothesis set is obtained.

The concept of generating correlation coefficients between the primary metric and a plurality of collected metrics may be further expanded, as in FIG. **17**, to generate correlation coefficients between several of the collected metrics and some or all other ones of the collected metrics, as in block **1704**. A hypothesis set may be generated based on these correlation coefficients, as in block **1705**. Additionally, correlation coefficients may be generated between the primary metric and all other collected metrics. Moreover, although requiring a substantial number of calculations, an all-pairs comparison for all the collected metrics in the application may be generated in order to discover relationships that may not have been evident when limiting the number of metrics that are correlated. An all-pairs comparison may include correlating all of the collected metrics with all others of the collected metrics.

Similar to the temporal shifting in the embodiment described in FIG. **15**, FIG. **17** illustrates embodiments in different user input types may be used to control temporal shifting during correlation analysis (blocks **1704** and **1705**). Similar to a previously described embodiment, a user interface may be used to receive an increment and/or decrement input (block **1701**), and adjust the time interval based on the received increment and/or decrement (block **1702**). In some embodiments, a temporal shift value may be received from the user interface, as in block **1703**, which may be used to adjust the time interval. Based on the time interval, the data corresponding to the collected metric corresponds to a first time while the data corresponding to the ones of the collected metrics being correlated corresponds to a second time that is different from the first time by the user specified time interval.

The correlation analyzer may be launched from the model generated by a health data processing application, as shown in FIG. **7**. A user may select a collected metric of interest on which to perform correlation analysis. For example, a user may click on an abnormal application response time metric to launch correlation analysis to determine causes for the poor performance. As another example, the user may click on an abnormal infrastructure metric to find out which higher-level function may affect it or be affected by it.

The information relating to the correlation analyzer may be presented to the user on a display, using graphical, textual, and/or tabular representations, as appropriate. Reference is made to FIG. **18** which illustrates a screen shot of a graphical user interface (GUI) including selection options for scope **1801** and network metric types **1802** for correlation analysis and display of results **1804** related to a primary metric by a correlation analysis application, according to some embodiments of the present invention. Information regarding the primary metric (also referenced as the correlated metric) **1803** may be displayed graphically, textually, and/or in tabular representation, among others. As shown by **1804**, for members of the candidate set that were correlated with the primary metric, the data may be displayed in tabular form. Information regarding the metric source, network element identification, context type, collected data, and correlation coefficient may be displayed. The displayed information may be sorted based on any of these types. Additionally, the candidate set and/or the hypothesis set may include visually distinctive display of members of the set by use of shading, boldface type, italicizing, fonts, colors, background colors, or other such ways to emphasize or highlight certain metrics. Additionally, distinctive highlighting may be used to distinguish the previously described thresholds relating to positive or negative correlations. Specific machines, processes, and/or links may be highlighted. Data graphs and/or sparklines may also be displayed to represent the primary metric and/or collected metrics. These sparklines and/or related information can be selected by the user to see larger displays with options to choose new correlation analysis parameters. The data may be presented statically or updated dynamically. Dynamically updated correlation data may be based on time intervals related to the sampling granularity of the data collection of the system, or based on update intervals selected by the user.

Reference is made to FIG. **19**, which illustrates a screen shot of a graphical user interface (GUI) including an example selection of transaction as the network metric type for correlation analysis and display of results related to a primary metric by a correlation analysis application, according to some embodiments of the present invention. The filter block **1802** provides an option to further limit the metrics in the candidate set for which correlations will be calculated against the primary metric. The filter block **1802** can filter in multiple ways, including, but not limited to, exact match, substring, prefix, suffix, and/or regular expression, among others.

Tracing and Aggregating Transaction Data

As described above, the collector application **200** generates performance data related to and/or indicative of the health of the network. Each collector application **200** may be collecting one machine's view of the application server. A large amount of data may be collected, including transaction specific information. Users that are presented with this voluminous data may desire to identify transactions that are being processed in this network, and/or events related to the transactions. In some embodiments, it may be assumed that all of the activity within an application happening at roughly the same time as a transaction is executing may be related to the transaction. However, a less inclusive approach may be desired.

Analysis showing individual transaction dependencies based on individual event dependencies may be referred to as front-to-back stitching. One goal of front-to-back stitching is to identify transaction dependencies by joining related events end-to-end. Front-to-back stitching may be used effectively in systems with thread exclusive behavior. In non-thread exclusive systems, time-based correlation, as described previously, may be an option. In some embodiments, techniques referred to as back-to-front stitching may be used to join multiple transaction instances together that include events that match time stamps and/or protocols.

Reference is made to FIG. **20**, which illustrates example operations carried out by a health data processing application to generate an aggregated list of transactions based on the thread identifier and transaction type, according to some embodiments of the present invention. Such transactions may occur between network applications and/or as a result of user input. Transactions may result in communication with one or more processes, which in turn may include one or more threads. Threads may be identified based on a thread identifier.

Some embodiments of the present invention apply to servers exhibiting thread exclusive behavior. Many systems provide thread exclusive behavior such that once a process starts working on a request using a thread, the thread works on nothing else until completion of the request. Thread exclusivity may occur on many servers that are extensible, where the server may hand off the thread to third party code. Examples of servers that exhibit thread exclusive behavior may include the Apache web server, IBM WebSphere, Oracle WebLogic Server, Sun ONE server, JBoss Application Server, Apache Tomcat, Caucho Resin, and/or other Java servers, among others. Other platforms for which some embodiments of the present application may be applied include .NET application servers, PHP, and/or Ruby on Rails, among others. Some embodiments of the present invention may apply to application servers for which thread identifiers are valid join identifiers. In some embodiments where the thread identifier is not a valid join identifier, events may be recorded when a transaction's flow of control switches from one thread identifier to another.

Referring to FIG. **20** at block **2001**, a thread identifier that is associated with a received transaction may be identified. The thread identifier used by a transaction for joining a thread may be unique system-wide and/or with respect to the application server on which it operates. Many operating systems provide thread identifiers that may be unique within a process, but not unique system-wide. For example, a first thread in each process may be labeled as "1" and may be unique when taken in conjunction with the process identifier. However, each thread may be associated with a control block in the kernel which may be related to a fixed location in memory for the life of the thread. The memory location and/or offset in memory of the control block may be used as a thread identifier.

As illustrated in block **2002**, once the thread identifier associated with the transaction is identified, events that are related to the thread identifier may be identified. Identified events may include calls associated with or resulting from the transaction. Identified events may also include transaction calls necessary to obtain data affiliated with the transaction.

In some embodiments, the type of the transaction may be determined, as illustrated in block **2003**. The type of the transaction may be a classification that could be used to group similar transactions for purposes of data aggregation. For example, types of transactions may include login, authenticate, read, write, connect, failure to connect, and/or logout, among others. Determining types of transactions resulting from certain operations may be useful to an operator in determining which types of transactions require more resources, cause more network problems, or result in error conditions.

As illustrated in block **2004**, the type of the transaction may be compared with the respective types of transactions among entries in the aggregated list of transactions. In some embodiments, an aggregated list of transactions may be maintained. The entries in the aggregated list may be stored in a variety of forms including hash tables, trees, linked lists, dynamic linked lists, pointer lists, and/or queues. The entries in the aggregated list of transactions may be organized in a variety of ways, and the aggregated list may be sorted in a number of convenient ways to improve computational efficiency. Examples of aggregated list sorting may include bubble sort, insertion sort, merge sort, quick sort, and/or comparison sort, among others.

As illustrated in block **2005**, the type of the transaction may be matched with the types of the entries in the aggregated list of transactions. In some embodiments, if the type of the transaction matches an entry in the aggregated list, the transaction and the one or more events related to the thread identifier related with the transaction are associated with the matching entry in the aggregated list of transactions based on. In some embodiments, the aggregated list of transactions may include one or more transactions received by a web or an application server in a time window.

Reference is now made to FIG. **21**, which illustrates example operations to identify transaction events based on networking protocols according to some embodiments of the present invention. As illustrated in block **2101**, events related to the thread identifier may be identified independent of the transmission protocol. In other words, events related to protocols such as HTTP, SQL, LDAP, TCP and/or others used by the transaction may be identified based on the thread identifier without consideration of their respective protocols.

According to some embodiments, block **2102** of illustrates selecting one or more protocols for use in identifying events related to the thread identifier. Protocols such as HTTP, SQL, LDAP, TCP and/or others that may be used by the transaction and/or related events may be selected. Selection may be based on user input, profiles, transaction type, and/or one or more of the related events.

As illustrated in block **2103**, events using the selected protocols may be identified as related to the thread identifier, whereas other events using non-selected protocols may not be identified as related to the thread identifier, even if they share a common thread identifier. In some embodiments, the protocols may be selected based on the received transaction. For example, if a login transaction is received, the HTTP protocol may be selected. Events of a given protocol may be associated with sequence numbers that may be used to determine event relationships based on the protocol. In some embodiments, a protocol specific identifier may be utilized to determine event relationships. For example, when using the HTTP protocol, HTTP protocol request headers may be used.

Reference is now made to FIG. **22**, which illustrates example operations to generate temporal performance data based on transaction and event times according to some embodiments of the present invention. According to block **2201**, a start time associated with a start of the transaction may be determined. This start time may be an absolute system time or relative measurements based on other events or on a present time window. The start time may correspond to a receipt of a request for the transaction. The start time may be obtained from time stamps in the header, trailer and/or other portions of data packets. The start time may be read from a system clock upon receipt of a request or event.

As illustrated in block **2202**, the stop time associated with the ending of a transaction may be determined. Similarly, the stop time may be an absolute system time or relative measurements based on other events or on a present time window. The stop time may correspond to completion of a request for a transaction, a return of operational functionality to a calling function associated with the transaction or cleanup of a process and/or thread associated with the transaction. The stop time may be obtained from time stamps in packets in the system. The stop time may be read from a system clock upon completion of a request or event.

According to some embodiments, as illustrated in block **2203**, an event time for respective ones of the one or more events related to the thread identifier associated with the transaction may be determined. The event time may be related to the start of an event, a completion of an event, or some time during the occurrence and/or execution of the related event. The event time may be obtained from time stamps in packets in the system. The event time may be read from a system clock at the start, completion, and/or some other time related to an event. The granularity of the determination of the event time may be less relevant since the event may be related by thread identifier to a transaction. Some embodiments provide that the relation to the start time and stop time of the transaction may be a more relevant parameter.

In some embodiments, it may be desired to identify events that take too long to complete, thereby slowing the related transaction. A difference between the stop transaction time and the start transaction time may be compared to a threshold to yield a determination that a transaction takes too long to complete. The threshold may be pre-defined and/or may be dynamically defined based on other available data and/or inputs. For example, transactions whose length of operation are greater than a threshold may be identified as taking too long and may be highlighted to the user. Additionally, this concept may be analogously extended to include determining start and stop times for events and highlighting events that may be taking too long to operate or complete.

As illustrated in block **2204**, temporal performance data corresponding to the transaction may be generated based on the start transaction time, the stop transaction time, and/or the event time. The temporal performance data may include statistical data. Statistical data may highlight types of transactions and their respective behavior by using statistical functions such as means, variance, distributions, co-variance, and/or probability functions, among others.

Reference is now made to FIG. **23**, which illustrates example operations carried out to identify and serialize events based on the thread identifiers of transactions according to some embodiments of the present invention. This trace mode of operation may facilitate tracing of specific transactions. At block **2301**, thread identifiers associated with corresponding received transactions may be identified. For received transactions, the respective thread identifiers of the plurality of threads may be unique in a web or an application server with respect to a kernel. The thread identifiers may be represented by an address location in the kernel of a control block of the thread. The thread identifiers may be represented by a pointer to an address location in the kernel of a control block of the thread.

At block **2302**, one or more events may be identified which correspond to the respective thread identifiers. Identified events may include calls associated with or resulting from the transaction. Identified events may also include transaction calls necessary to obtain data affiliated with the transaction.

Determining that some events or some back-end behavior is related to a given user request may be sensitive to ordering. For example, a back-end event or activity starts at time $T_1$ and a user request arrives at time $T_2$, which is after time $T_1$ In this example, the back-end event or activity may not be caused by the later arriving user request since causal events may be linked forward in time. According to block **2303**, the events related to the plurality of thread identifiers may be serialized into a serialized list of events. In some embodiments the list of events may be an order list.

Preserving the order of events running in different contexts may be difficult. Reference is now made to FIG. **25**, which illustrates example operations to serialize events as in block **2303**, by passing events through a single kernel queue and/or using a spinlock according to some embodiments of the present invention. As illustrated in block **2501**, serializing the events may include passing the events though a single kernel queue. In some embodiments, serializing the events may include using a spinlock that imposes an order of events in the single kernel queue, as illustrated in block **2502**. In this case, an event that happened at time $T_1$ may be placed in the kernel queue before an event at that happened at a later time $T_2$. This technique may include some processor performance penalty. In some embodiments, to reduce the performance penalty in systems with an available monotonic (always increasing) clock, events could be recorded with timestamps without using a spinlock. These events, based on the timestamps could be reordered outside of the driver.

Reference is made to FIG. **24**, which illustrates example operations to identify transaction events based on networking protocols according to some embodiments of the present invention. As shown in block **2302**, events may be identified that correspond to thread identifiers. According to block **2401**, events related to the thread identifier may be identified independent of the transmission protocol. In other words, events related to protocols such as HTTP, SQL, LDAP, TCP and/or others used by the transaction may be identified based on the thread identifier without consideration of their respective protocols.

According to some embodiments, block **2402** illustrates selecting one or more protocols for use in identifying events related to the thread identifier. Protocols such as HTTP, SQL, LDAP, TCP and/or others used by the transaction may be selected. Selection may be based on user input, profiles, transaction type, and/or one or more of the related events.

As illustrated in block **2403**, events using the selected protocols may be identified as related to the thread identifier, whereas other events using non-selected protocols may not be identified as related to the thread identifier, even if they share a common thread identifier. In some embodiments, the protocols may be selected based on the received transaction. For example, if a login transaction is received, the HTTP protocol may be selected. In some embodiments, a protocol specific identifier may be utilized to determine event relationships. For example, when using the HTTP protocol, the HTTP protocol client header may be used.

Reference is now made to FIG. **26**, which illustrates tracing mode behavior by assigning a stitching identifier and maintaining a stitching counter for events based on the thread identifier according to some embodiments of the present invention. One server and/or collector's view of a transaction may be referred to as a shard. A transaction may touch several servers, but a shard is one server and/or collector's view of the transaction. A shard instance may include a piece of a transaction. A shard pattern may be a union of events that look the same in a time window. For example, it may be desired to analyze multiple users loading their respective shopping carts. Whenever users load a shopping cart, several back-end events may occur. Loading one user's shopping cart may result in accessing a database while loading a different user's shopping cart may result in accessing an authentication server. The shard pattern for this example is the union of events including events such as accessing the database and accessing the authentication server. This shard pattern is a summarization of events that may be valuable for an overall view of the collector's behavior. Additionally, an increased granularity of information may be achieved by viewing these shard instances in a tracing mode to view behavior of individual transactions. Shard instances and patterns may be presented to the user in a variety of ways including tables and heat maps. Tracing mode behavior that analyzes individual transaction dependencies based on individual event dependencies may be referred to as front-to-back stitching. Front-to-back stitching may identify transaction dependencies by joining related events end-to-end.

As illustrated in block **2601** of FIG. **26**, a stitching identifier may be associated with a received transaction and/or events. The stitching identifier may be assigned by the collector application and may be unrelated to the thread identifier. The stitching identifier may be used to track input and output events related to the transaction.

As illustrated in block **2602**, the stitching identifier may be assigned to events corresponding to a thread identifier. For example, an input account login transaction may be assigned a stitching identifier "5". A login event may be triggered by the received login transaction. Events related to this login transaction such as the login event, the retrieveUserName event and the authenticate event may also be assigned a stitching identifier "5".

As illustrated in block **2603**, a stitching counter may be associated with the stitching identifier. The stitching counter may count events associated with the stitching identifier. The stitching counter may be incremented for each of the one or more events assigned to the stitching identifier. In the above example, the login event may increment the stitching count to a value of "1". The retrieveUserName event may increment the stitching count to "2" and the authenticate event may increment the stitching count to a value of "3".

Reference is now made to FIG. **27**, which illustrates determining and counting events for a transaction to be traced according to some embodiments of the present invention. The above example may be continued for illustrative purposes. As illustrated in block **2701**, a tracing transaction (i.e. a transaction to be traced) may be identified. In this example the login transaction may be identified as the tracing transaction.

As shown in block **2702**, a trace stitching identifier associated with the tracing transaction may be identified. In this foregoing example, the trace stitching identifier is "5". As illustrated in block **2703**, one or more events corresponding to the trace stitching identifier may be determined from the serialized list of events. In the foregoing example, events corresponding to a stitching identifier of "5" may be determined. The events in this example may include the retrieveUserName event and the authenticate event, since these events have been assigned a stitching value of "5". These events may correspond to the tracing transaction.

According to block **2704**, a total event counter may be incremented for ones of the one or more events corresponding to the trace stitching identifier in order to generate a total event count value. In the foregoing example, the total event counter may be incremented for each of the login, retrieveUserName, and authenticate events, such that the total event counter may have a value of "3".

Reference is now made to FIG. **28**, which illustrates generating indicators based on front-to-back stitching according to some embodiments of the present invention. As shown in block **2801**, if no accurate count can be calculated, a counter value of "0" can be used to indicate "no data available.". The total event counter may be equal to 0 if there are null and/or "0" stitching identifiers and null and/or "0" stitching counts.

According to some embodiments as shown in block **2802**, if a transaction event is not associated with any other events, a counter value of "1" can be used to indicate that there are "no causal events." If the total event counter value is equal to 1, no other events may be related to the single event triggered by the transaction request. In the previously discussed example, if the login transaction triggered only the login event and no other subsequent events, the total event counter may be equal to 1.

As illustrated in block **2803**, the stitching counter may be compared to the total event counter value. A "missing events" indicator may be generated if there is a mismatch between the stitching counter value and total event counter value. These values may not match if there is a restriction on the rate, number, or kind of trace events that may be recorded or reported by a collector application **200**. Knowledge of missing events may help indicate reasons transactions are not completing, timing out, event deadlocks, and/or other undesired behavior in the system.

Transaction Relationships Between Application Servers

Analysis showing transaction dependencies between web or application servers may be referred to as back-to-front stitching. One goal of back-to-front stitching is to identify end-to-end transaction dependencies spanning multiple application servers in a system. Back-to-front stitching may be used effectively in systems where relationships between outbound transactions and inbound transactions across web or application servers may be discovered.

Reference is now made to FIG. **29**, which illustrates a trace mode based on back-to-front stitching, according to some embodiments of the present invention. As illustrated in block **2901**, an outbound transaction may be identified. The outbound transaction may be a transaction from one web or application server to another web or application server. The outbound transaction may trigger other web or application servers to create other transactions. For example, an Apache server may send an authenticate transaction to a WebLogic server. This transaction is recorded as an outbound transaction by the Apache server and also as an inbound transaction by the WebLogic server.

Block **2902** illustrates that a transaction type may be identified for an outbound transaction. The type of the transaction may be a classification that could be used to relate inbound and outbound transactions for purposes of trace mode data collection. For example, types of transactions may include login, authenticate, read, write, connect, failure to connect, and/or logout, among others. Determining types of transactions resulting from certain operations may be useful to an operator in determining which types of transactions cause dependencies between certain application servers. These dependencies may require more communication bandwidth between application servers, processor resources, and/or memory resources, among others. They may also cause undesired behavior, including timeouts, deadlocks, or errors, among others.

According to some embodiments, candidate transactions may be identified (block **2903**). The candidate transactions may be inbound transactions that have the same transaction type as the transaction type of the outbound transaction. In some embodiments, an assumption may be made that related inbound and outbound transactions may have the same transaction type. In other embodiments, transaction types suspected of having causal relationships may be grouped together. These groups of transactions may be included in the basis for determining candidate transactions.

As illustrated in block **2909**, the inbound transactions identified as candidate transactions may occur in a given time window. The inbound transactions identified as candidate transactions may have latency approximately equal to the latency of related outbound transactions.

According to some embodiments, one or more transaction data parameters that correspond to the outbound and/or inbound transactions block may be selected (**2904**). A candidate transaction may be determined to be related to the outbound transaction based on the one or more transaction data parameters. Transaction data parameters may be protocol independent and/or protocol dependent. According to some embodiments, the one or more transaction data parameters may include a protocol specific identifier. The protocol specific identifier may include a header. For example, Hypertext Transfer Protocol (HTTP) request headers may be included as a transaction data parameter.

As illustrated in block **2905**, a non-reversible checksum may be generated based on at least a portion of the header. For security purposes, privacy preserving techniques to generate a non-reversible checksum may be employed. To generate a non-reversible checksum, some embodiments provide that only a portion of the header may be used in calculating the checksum. In some embodiments, the checksum may be implemented using a hash function, checksum, or cyclic redundancy check, among others.

Additionally, as illustrated in block **2906**, non-reversible checksums may be obtained by generating the checksum on non-contiguous portions of the header. In a non-limiting example, every other bit of the header may be used for generation of the checksum. The checksum size may be adjusted to be shorter or longer, depending on the granularity needed to determine a somewhat unique transaction data parameter.

According to some embodiments, transaction data parameters may include a query string. As illustrated in block **2907**, a non-reversible checksum may be generated based on at least a portion of the query string. For security purposes, privacy preserving techniques to generate a non-reversible checksum may be employed. To generate a non-reversible checksum, some embodiments provide that only a portion of the query string may be used. For example, the first 2048 bytes of the query string may be used to generate the checksum. A portion of the query string may be used to reduce the computational effort necessary to generate the checksum. A portion of the query string may be used in cases where less than the entire query string is available for computing a checksum.

Additionally, as illustrated in block **2908**, non-reversible checksums may be obtained by generating the checksum on non-contiguous portions of the query string. In a non-limiting example, every other bit and/or byte of the query string may be used for generation of the checksum. The checksum size may be adjusted to be shorter or longer, depending on the granularity needed to determine a somewhat unique transaction data parameter.

Reference is now made to FIG. **30**, which illustrates selecting transaction data parameters in trace mode using back-to-front stitching, according to some embodiments of the present invention. As illustrated in block **2904**, transaction data parameters may be selected. According to block **3001**, some protocols include a transaction identifier, which may be identified to be one of the transaction data parameters. In these cases, the protocols tag each transaction with a unique identifier which may be used to determine relationships between inbound and outbound transactions on different applications servers.

As illustrated in block **3002**, some protocols include an error code and/or a return code, which may be identified to be one of the transaction data parameters. The error code and/or return code generated as a transaction is processed by an application server may be useful to identify relationships between inbound and outbound transactions. Specifically, the error and/or return code recorded on an outbound view of a transaction may match the error and/or return code recorded on an inbound view of the same transaction.

According to some embodiments, block **3003** illustrates that a port number and/or a sequence number may be identified as one of the transaction data parameters. For example, the sequence number may be related to a TCP data stream. The sequence number may be determined based on a byte offset of the TCP data stream.

Reference is now made to FIG. **31**, which illustrates an aggregate mode using back-to-front stitching according to some embodiments of the present invention. At block **3101**, a transaction type may be identified for analysis of one or more application servers. Network operators may desire to understand the aggregate behavior of many instances of a certain transaction type. For example, the operator may select the login transaction type to determine system behavior for multiple login transactions occurring in a time window.

According to some embodiments, block **3102** illustrates identifying outbound transactions of the identified transaction type. For example, all outbound transactions from application servers of transaction type login may be determined.

In some embodiments, as illustrated at block **3103**, a plurality of inbound transactions of the identified transaction type may be determined. In the foregoing example, the inbound login transactions may occur on many different application servers. The inbound login transactions may be related to one or more the outbound login transactions of some of the application servers.

In some embodiments, as illustrated in block **3104**, information associated with the plurality of outbound transactions and information associated with the plurality of inbound transactions of a same transaction type may be aggregated. These relationships may be highlighted to the network operation in order to establish overall system dependencies between various application servers in the network.

In some embodiments, as illustrated at block **3105**, the protocol and/or port number associated with the outbound transactions may be identified. For example, outbound login transactions may be associated with HTTP protocol login requests. Outbound port numbers for login transactions may be identified.

As shown in block **3106**, a plurality of related inbound transactions for one or more different application servers may be identified that are based on the protocol and/or the port number associated with the plurality of outbound transactions. In some embodiments, these identified inbound transactions may be of the same transaction type as the related outbound transactions. In other words, any combination of transaction type, protocol, and/or port number may be used to determine inbound transactions related to outbound transactions.

Reference is made to FIG. **32**, which illustrates a screen shot of a graphical user interface (GUI) including example tracing of transaction events according to some embodiments of the present invention. Block **3201** illustrates the starting of a time window in which traffic data associated with an orderQuery transaction may be analyzed. In this example, the start time, response time, client IP address, and a node determined to be a bottleneck are identified.

Block **3202** may represent a snapshot of the transactions and events occurring in the system. An orderQuery transaction is traced in this example. Various servers such as an Apache HTTP Server, a Sun ONE Web Server, and a WebSphere server, each with delays, are illustrated in this example. The WebSphere server is further illustrated to include Port 9000, which has a delay of 327 ms and IBM®DB2®, which has a delay of 828 ms.

Block **3203** may represent traces that illustrate events related to a specific transaction and associated response times. In this example, an orderQuery transaction may result in events such as a "SELECT * FROM ORDERS" event and a "SET CLIENT WRKSTNNAME" event. The response times of these events (828 ms each) are shown in the table along with a transaction response time of 6,245 ms.

Many variations and modifications can be made to the embodiments without substantially departing from the principles of the present invention. The following claims are provided to ensure that the present application meets all statutory requirements as a priority application in all jurisdictions and shall not be construed as setting forth the scope of the present invention.

That which is claimed:

**1**. A method of analyzing network traffic comprising:

identifying a thread identifier associated with a transaction;

identifying one or more events related to the thread identifier;

determining a type of the transaction to be a first type of transaction;

comparing the first type of transaction with respective types of transactions among entries in an aggregated list of transactions; and

associating the transaction and the one or more events related to the thread identifier associated with the transaction with an entry in the aggregated list of transactions based on a match between the first type of transaction and a respective one of the types of transactions associated with the entry in the aggregated list of transactions, wherein the thread identifier is represented by an address location and/or a pointer to an address location in a kernel of a control block of a thread corresponding to the thread identifier.

**2**. The method of analyzing network traffic of claim **1**, wherein the aggregated list of transactions comprises one or more transactions received by an application server in a time window.

**3**. The method of analyzing network traffic of claim **1**, further comprising:

determining a start transaction time associated with a start of the transaction;

determining a stop transaction time associated with completion of the transaction; and

determining an event time for respective ones of the one or more events related to the thread identifier associated with the transaction.

**4**. The method of analyzing network traffic of claim **3**, wherein the start of the transaction corresponds to a receipt of a request for the transaction.

**5**. The method of analyzing network traffic of claim **3**, wherein a difference between the stop transaction time and the start transaction time is greater than a threshold.

**6**. The method of analyzing network traffic of claim **3**, further comprising:

generating temporal performance data corresponding to the transaction based on the start transaction time, the stop transaction time, and/or the event time.

**7**. The method of analyzing network traffic of claim **6**, wherein the temporal performance data comprises statistical data.

**8**. The method of analyzing network traffic of claim **1**, wherein the method of claim **1** is applied to a web or an application server determined to use thread identifiers as valid join identifiers.

**9**. The method of analyzing network traffic of claim **8**, wherein the web or the application server includes an Apache server, a Sun ONE server and/or a Java-based application server.

**10**. The method of analyzing network traffic of claim **8**, wherein the thread identifier is unique among threads across the application server.

**11**. The method of analyzing network traffic of claim **1**, wherein identifying the one or more events comprises identifying one or more events that include any of the one or more protocols used by the respective transaction.

**12**. The method of analyzing network traffic of claim **1**, wherein identifying the one or more events comprises:

selecting one or more protocols; and

identifying one or more events associated with the one or more selected protocols.

**13**. The method of analyzing network traffic of claim **12**, wherein the one or more selected protocols are associated with the transaction.

**14**. A method of analyzing network traffic comprising:

identifying a plurality of thread identifiers, ones of which are associated with corresponding ones of a plurality of received transactions;

identifying one or more events corresponding to the respective ones of the plurality of thread identifiers; and

serializing the one or more events related to the plurality of thread identifiers into a serialized list of events,

wherein at least one of the thread identifiers is represented by an address location and/or a pointer to an address location in the kernel of a control block of the thread.

**15**. A method of analyzing network traffic comprising:

identifying a plurality of thread identifiers, ones of which are associated with corresponding ones of a plurality of received transactions;

identifying one or more events corresponding to the respective ones of the plurality of thread identifiers; and

serializing the one or more events related to the plurality of thread identifiers into a serialized list of events, for ones of the plurality of received transactions, further comprising:

associating a stitching identifier with a transaction of the plurality of received transactions;

assigning the stitching identifier to the one or more events corresponding to one of the plurality of thread identifiers;

associating a stitching counter with the one of the plurality of thread identifiers that counts events associated with the stitching identifier; and

incrementing the stitching counter for each of the one or more events assigned to the stitching identifier.

**16**. The method of analyzing network traffic of claim **15**, further comprising:

identifying a tracing transaction;

identifying a trace stitching identifier that is associated with the tracing transaction; and

determining, from the serialized list of events, one or more events corresponding to the trace stitching identifier to identify ones of the one of more events that correspond to the tracing transaction.

**17**. The method of analyzing network traffic of claim **16**, further comprising:

incrementing a total event counter for ones of the one or more events corresponding to the trace stitching identifier to generate a total event count value.

**18**. The method of analyzing network traffic of claim **17**, further comprising:

generating a no data available indicator based on the total event counter value equal to 0.

**19**. The method of analyzing network traffic of claim **17**, further comprising:

generating a no causal events indicator based on the total event counter value equal to 1.

**20**. The method of analyzing network traffic of claim **17**, further comprising:

comparing the stitching counter associated with the trace stitching identifier with the total event counter; and

generating a missing events indicator based on a mismatch between the stitching counter associated with the trace stitching identifier and the total event counter.

**21**. The method of analyzing network traffic of claim **14**, wherein the respective thread identifiers of the plurality of threads are unique in an application server with respect to a kernel.

**22**. The method of analyzing network traffic of claim **14**, wherein serializing the one or more events comprises passing events through a single kernel queue.

**23**. The method of analyzing network traffic of claim **22**, wherein serializing the one or more events further comprises using a spinlock that imposes an order of events in the single kernel queue.

**24**. The method of analyzing network traffic of claim **14**, wherein identifying the one or more events comprises identifying one or more events that include any of the one or more protocols used by the corresponding one of the received transactions.

**25**. The method of analyzing network traffic of claim **14**, wherein identifying the one or more events comprises:

selecting one or more protocols; and

identifying one or more events associated with one or more selected protocols used by the corresponding one of the received transactions.

**26**. The method of analyzing network traffic of claim **25**, wherein the one or more selected protocols are associated with one of the received transactions.

**27**. A method of analyzing network traffic comprising:

identifying one or more events that are associated with a transaction based on a thread identifier; and

associating the transaction and the one or more events with at least one transaction of a plurality of transactions based on a transaction type,

wherein the thread identifier is represented by an address location and/or a pointer to an address location in a kernel of a control block of a thread corresponding to the thread identifier.

**28**. The method of analyzing network traffic of claim **27**, further comprising:

determining a start transaction time associated with a start of the transaction;

determining a stop transaction time associated with completion of the transaction; and

determining an event time for respective ones of the one or more events related to the thread identifier associated with the transaction.

**29**. The method of analyzing network traffic of claim **28**, wherein a difference between the stop transaction time and the start transaction time is greater than a threshold.

**30**. The method of analyzing network traffic of claim **28**,

wherein the start transaction time, the stop transaction time, and/or the event time are used to generate temporal performance data corresponding to the transaction, and

wherein the temporal performance data comprises statistical data.

**31**. The method of analyzing network traffic of claim **27**, wherein identifying the one or more events comprises identifying one or more events that include any of the one or more protocols used by the respective transaction.

**32**. The method of analyzing network traffic of claim **27**, wherein identifying the one or more events comprises:

selecting one or more protocols; and

identifying one or more events associated with the one or more selected protocols,

wherein the one or more selected protocols are associated with the transaction.

**33**. A computer program product, the computer program product comprising a non-transitory computer usable storage medium having computer-readable program code embodied in the medium, the computer readable program code configured to perform the method of claim **1**.

**34**. A computer program product, the computer program product comprising a non-transitory computer usable storage medium having computer-readable program code embodied in the medium, the computer readable program code configured to perform the method of claim **14**.

* * * * *